UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

DEFINIÇÃO DAS PROBABILIDADES CONDICIONAIS DE REDES BAYESIANAS BASEADAS EM NÓS RANQUEADOS

RAISSA MATIAS DA SILVA

HYGGO ALMEIDA
ANGELO PERKUSICH
(ORIENTADORES)

CAMPINA GRANDE
AGOSTO - 2016

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Ciência da Computação

Definição das Probabilidades Condicionais de Redes Bayesianas baseadas em Nós Ranqueados

Raissa Matias da Silva

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Angelo Perkusich
Hyggo Almeida
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586d Silva, Raissa Matias da.
Definição das probabilidades condicionais de redes bayesianas baseadas em nós ranqueados / Raissa Matias da Silva. – Campina Grande, 2016.
138 f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2016.
"Orientação: Prof. Dr. Angelo Perkusich, Prof. Dr. Hyggo Almeida.
Referências.

1. Redes Bayesianas. 2. Tabela de Probabilidade Condicional. 3. Processo de Desenvolvimento de Software. 4. Sistema Especialista. 5. Engenharia de Software. I. Perkusich, Angelo. II. Almeida, Hyggo. III.Título.

CDU 004.41(043)

Resumo

Um dos principais desafios na construção de uma Rede Bayesiana (RB) é definir as tabelas de probabilidade condicional dos nós (TPC). Para RB de larga escala, aprender TPC por meio da elicitação de domínio do conhecimento de um especialista é inviável. Trabalhos anteriores propuseram soluções para este problema usando o conceito de nós ranqueados, no entanto, eles têm capacidade limitada de modelagem ou precisam contar com especialistas em RB parar aplicá-los, reduzindo a sua aplicabilidade. Neste trabalho, são propostos e avaliados três métodos para resolução deste problema. O primeiro utiliza um sistema especialista baseado em regras de produção. O segundo método utiliza força bruta, buscando um conjunto de todas as combinações possíveis. O terceiro método utiliza um algoritmo genético para definição de TPC por especialistas sem conhecimento específico de nós ranqueados. Para avaliar as abordagens, foi executado um experimento que permitiu identificar as vantagens e as desvantagens de cada método, dependendo do tempo de processamento, disponibilidade de memória e a quantidade de nós pais da RB. Ao usar alguma das soluções apresentadas, um praticante pode definir com maior precisão as TPC sem entender o conceito de nós ranqueados.

Palavras-Chave: Redes Bayesianas; Tabela de Probabilidade Condicional; Processo de Desenvolvimento de Software; Sistema Especialista.

Abstract

One of the key challenges in constructing a Bayesian network (BN) is defining the node probability tables (NPT). For large-scale BN, learning NPT through domain experts knowledge elicitation is unfeasible. Previous works proposed solutions to this problem using the concept of ranked nodes; however, they have limited modeling capabilities or rely on BN experts to apply them, reducing their applicability. In this work, we propose and evaluate three methods to solve the problem. First, an expert system based on production rules. Second, a method using a brute-force algorithm to identify a set of possible combination. Finally, a method using genetic algorithm to define NPTs with no ranked nodes-specific knowledge. To validate this approach, it was executed an experiment with a BN already published in the literature. Results demonstrated the advantages and disadvantages of each method depending on time, memory availability and parents node quantity. By using one of the presented solution, a practitioner can accurately define NPTs without understanding the concept of ranked nodes.

Agradecimentos

Agradeço primeiramente à Deus, por me conceder saúde e sabedoria para concluir mais uma etapa na minha vida. A realização deste trabalho não seria possível sem o apoio da minha família, agradeço imensamente aos meus pais Djalma Matias e Izaneide Fernandes que sempre me apoiaram, que foram os primeiros a garantir minha educação e é por eles que eu continuo essa batalha diária, eles que nunca deixaram faltar amor. Aos meus irmão Layza e Djalminha pelo companheirismo. Agradeço ao meu namorado Arthur, por todo o amor, paciência, dedicação e motivação durante todo esse processo.

Grande parte desta pesquisa se deve à orientação que obtive durante a sua realização. Agradeço ao meu orientador Mirko, por toda a paciência e dedicação na definição e conclusão desse trabalho. Agradeço também aos professores e orientadores Hyggo Almeida e Angelo Perkusich pela ajuda e aprendizado.

Não posso esquecer de agradecer aos meus colegas de trabalho, que todos os dias me cobraram atualizações do mestrado. Agradeço também aos meus amigos, que me incentivaram e me deram força para conclusão dessa fase.

Conteúdo

Lista de Símbolos

TPC - *Tabela de Probabilidade Condicional*

RB - *Rede Bayesiana*

Lista de Figuras

Capítulo 1

Introdução

Uma Rede Bayesiana (RB) é um grafo acíclico dirigido cujos nós representam variáveis aleatórias e arcos representam as dependências entre os mesmos. Redes Bayesianas são utilizadas para representar conhecimento incerto e fazer o usuário raciocinar sobre incerteza. Áreas de aplicação das mesmas incluem, por exemplo, diagnóstico médico [6], [23], [32], resolução de problemas de hardware e diagnóstico [7], problemas de planejamento de desenvolvimento de software [31], simulação e metamodelagem [38], [21], e planejamento militar [15], [37].

Os nós da RB podem ser representados em uma escala contínua ou discreta. Cada valor possível é chamado de estado. No caso de número finito de estados, as dependências são definidas por tabelas de probabilidade condicional (TPC). De acordo com Laitila *et al.* [27], uma TPC especifica a distribuição de probabilidade condicional do nó ascendente, o nó filho, para todas as combinações dos estados de seus antecessores diretos, os nós pai. A TPC e a estrutura gráfica da RB codificam a distribuição de probabilidade conjunta das variáveis aleatórias discretas. Quando são conhecidos os estados de alguns nós, as distribuições de probabilidades dos outros podem ser atualizados em conformidade usando algoritmos eficientes, tais como os apresentados em [43] e [44].

Portanto, RB é uma ferramenta eficaz para responder consultas probabilísticas sobre variáveis aleatórias e que proporcionam meios para realizar, por exemplo, análise de risco [45; 19]. Além disso, em vários cenários, Redes Bayesianas oferecem uma forma de apoiar a tomada de decisões em condições de incerteza [37; 24].

Segundo Perkusich *et al.* [34], existem duas barreiras significantes para a construção

de Redes Bayesianas de larga escala: contrução do gráfo acíclico dirigido e das TPC. Esse trabalho tem como foco a segunda barreira.

Definir TPC é um trabalho de esforço exponencial e esse tipo de estudo tem sido o foco de vários trabalhos passados. Existem duas formas de coletar dados para definir as TPC: por banco de dados ou elicitação de conhecimento de especialistas do domínio. A definição de TPC por banco de dados é automatizada por meio de um processo chamado *batch learning* [18]. Por outro lado, na prática, são raros os casos nos quais há um banco de dados adequado. Então, para definir as TPC, é necessário coletar dados de especialistas. O problema é que definir manualmente as TPC pode se tornar inviável dependendo do número de nós e estados. Como apresentado em Fenton *et al.* [31], inconsistências podem ocorrer se especialistas de domínio, exaustivamente, definirem as TPC de um nó composto de uma grande quantidade (e.g., 125) de estados.

Na ausência de dados, as TPC têm que ser construídas com base em levantamento envolvendo as avaliações subjetivas de um especialista de domínio. A principal dificuldade nisso é a grande quantidade de avaliações de probabilidade necessária. À medida que o tamanho de uma tabela de probabilidade condicional cresce exponencialmente com o número de nós pais, a quantidade de avaliações de probabilidade necessária para uma única rede Bayesiana pode ser de até centenas ou milhares. Avaliando tantas probabilidades de forma coerente e sem preconceitos pode ser um problema insuperável para o especialista, devido à tensão cognitiva ou escassez de tempo [11], [30]. Por exemplo, técnicas de probabilidade de elicitação convencionais, tais como rodas de probabilidade de loteria ou de referência, são geralmente reconhecidas como sendo muito demoradas para a elicitação da probabilidade de Redes Bayesianas [11], [30], [25]. O problema principal é, dado que o conhecimento de um especialista é elicitado, como automatizar a definição de nós ranqueados.

Na literatura há soluções propostas para reduzir a complexidade de definir TPC de larga escala. Exemplos de soluções são *Noisy-OR* [33], projetado para variáveis binárias, e *Noisy-MAX* [10], que é a generalização da primeira para variáveis com vários estados. *Noisy-OR* tem a desvantagem que se aplica apenas aos nós booleanos e implicitamente ignora os efeitos de interação entre as variáveis. *Noisy-MAX* é aplicado para escala ordinal, mas sua capacidade de modelagem é limitada [31].

Outra solução é apresentada em Fenton *et al.* [31]. A mesma é baseada no conceito de

nó ranqueado. Nós ranqueados representam variáveis discretas cujos estados são expressos em uma escala ordinal e podem ser mapeadas em uma escala numérica limitada, contínua e monotonamente ordenada no intervalo $[0, 1]$. Para definir as TPC, a solução utiliza expressões ponderadas. O uso das mesmas auxiliam os especialistas a compreender e expressar relações probabilísticas entre nós.

Um exemplo de utilização de nó ranqueado é apresentado em Perkusich *et al.* [35], no qual o mesmo foi aplicado para construir uma rede Bayesiana para detectar problemas em processos de software. A solução foi aplicada com sucesso em projetos de desenvolvimento de software baseado em *Scrum*, aumentando a eficiência dos processos de software e por consequência, aumentando as chances de sucesso dos projetos. Outros exemplos de aplicação de nós ranqueados são segurança [12], sistemas críticos de usina nuclear [17], segurança de laboratórios de química [8] e análise de risco em cadeia logística aeroespacial [40]. Em Laitila *et al.* [27] foi apresentado um método para utilizar nós ranqueados para escalas intervalares.

Visto que Redes Bayesianas e nós ranqueados são utilizados em várias aplicações, a construção da TPC torna-se fundamental para a confiabilidade que a Rede Bayesiana, que será apresentada, oferece. Para que a TPC seja gerada de forma mais precisa e confiável, a melhor função de probabilidade condicional precisa ser definida.

O método de Fenton *et al.* [31] baseia-se na utilização da Distribuição Normal Truncada (TNormal) para representar TPC. TNormal é utilizada devido a sua capacidade de assumir diversas formas e, assim, modelar diversos tipos de relacionamentos [31]. Para definir uma TNormal, é necessário definir quatro variáveis: μ, média; σ^2, variância de erro; a, limite inferior; e b, limite superior. Neste caso, dado que utiliza-se o intervalo [0 - 1] para representar os estados das variáveis, assume-se que o limite inferior a = 0 e o limite superior b = 1. μ (média), a qual representa a tendência central da distribuição, é definida por meio de funções de probabilidade paramétricas onde as probabilidades do nó filho são definidas de acordo com uma função ponderada dos valores dos nós pai. Há quatro tipos de funções: *Weighted Mean Average (WMEAN), Weighted Minimum (WMIN), Weighted Maximum (WMAX), Mix Minimum-Maximum (MIXMINMAX).* σ^2 é (variância) definida empiricamente e representa a confiança nas probabilidades calculadas. A mesma permite modelar a forma da curva, incluindo uma distribuição uniforme, alcançada quando σ^2 >= 8 (i.e., valor definido empiri-

camente por Fenton *et al.* [31] e enviesada quando $\sigma^2 = 0$). Segundo Fenton *et al.* [31], 93% de esforço foi reduzido quando comparou sua abordagem com uma abordagem manual.

1.1 Problemática

Por outro lado, o processo para definição de μ e σ^2 apresentado em Fenton *et al.* [31] é limitado e subjetivo, pois este depende do conhecimento e experiência do construtor da RB para definir o tipo de expressão ponderada, pesos e σ^2. Para tal, de acordo com Fenton *et al.* [31], o mesmo deve coletar dados do especialista por meio de valores de amostragem, que resultam em afirmações de elicitação de especialistas como as seguintes:

- Quando X1 e X2 são ambos "muito alto", a distribuição de Y é fortemente enviesada para "muito alto";
- Quando X1 e X2 são ambos "muito baixo", a distribuição de Y é fortemente enviesada para "muito baixo";
- Quando X1 é "muito baixo" e X2 é "muito alto", a distribuição de Y é centralizada abaixo de "médio";
- Quando X1 é "muito alto" e X2 é "muito baixo", a distribuição de Y é centralizada acima de "médio".

Uma vez que é assumido que cada nó tem uma escala numérica subjacente no intervalo [0, 1], tais afirmações sugerem intuitivamente que Y é algum tipo de função de média ponderada. Na verdade, os especialistas acham mais fácil de compreender e expressar as relações em tais termos.

Desta forma, conclui-se que a definição das TPC depende de conclusões subjetivas do projetista da Rede Bayesiana. Além disso, por não haver uma pré-definição dos valores de amostragem a serem coletados, pode-se concluir que o processo é *ad-hoc*. Desta forma, há riscos associados ao mesmo, tais como viés e erros humanos.

Perkusich *et al.* [34] complementam o trabalho de Fenton *et al.* [31] encapsulando a complexidade de definir μ e σ^2. Por outro lado, apresenta limitação da capacidade de modelagem. Para a definição de μ, Perkusich *et al.* [35] limita-se a utilizar apenas *WMEAN*.

Para definição de σ^2, definiu-se o valor fixo de $5.0E^{-4}$, que é valor mínimo permitido no AgenaRisk [1] (i.e., ferramenta utilizada para executar a Rede Bayesiana). Dado que as TPC quantificam relacionamentos, essas limitações podem diminuir a acurácia da quantificação.

O método proposto no trabalho de Perkusich *et al.* [34] consiste na coleta de informações de especialistas de domínio por meio de uma pesquisa, e, com os dados coletados, construir TPC da RB. Em outro trabalho, Perkusich *et al.* [35] diminuem a participação do segundo especialista, porém, para o cálculo das tabelas de probabilidade da Rede Bayesiana em questão, o método utilizado considera apenas a função WMEAN, como mencionado anteriormente, o que para alguns casos, pode não ser a mais adequada.

Diante dessas limitações, procurou-se, neste trabalho, encapsular o conhecimento de baixo nível necessário para utilizar nós ranqueados diminuindo os problemas apresentados. Com a elicitação do especialista para a análise das funções, é indicado ao usuário a melhor função para o cálculo das TPC, considerando as funções WMEAN, WMAX, WMIN e MIXMINMAX. Desta forma, torna-se o método de análise do desenvolvimento do processo de software mais acessível (método de validação), uma vez que será necessário apenas um especialista, que seria o *Scrum Master*, além de torná-lo mais confiável, uma vez que existe a possibilidade de escolha de uma função mais apropriada para a definição das TPC.

Desta forma, pode-se concluir que não há uma solução consolidada para viabilizar a utilização de nó ranqueado em contextos nos quais não há um especialista no método para calibrar a TNormal, o que limita sua aplicabilidade. Neste contexto, define-se o problema técnico desta pesquisa: dado que o conhecimento de um especialista é elicitado, como automatizar a definição de nós ranqueados encapsulando conhecimentos de TNormal?

1.2 Objetivos

O objetivo principal neste trabalho é propor soluções para automatizar a definição, de acordo com dados coletados de um especialista, dos parâmetros (tipo de função, pesos e variância) das funções de probabilidade de um nó ranqueado. Desta forma, encapsula-se do usuário de nós ranqueados o conhecimento técnico em TNormal necessário para utilizá-los. Com esse propósito, foram desenvolvidos três métodos, cada um com suas vantagens e desvantagens.

Para a quantificação dos relacionamentos da Rede Bayesiana, foi realizado um estudo

empírico com combinações de perguntas e respostas utilizando o *AgenaRisk*. A validação e estudos de caso foram realizados em projetos reais em desenvolvimento.

Desta forma, definem-se os seguintes objetivos específicos:

- Definição de métodos para construir TPC automaticamente;
- Desenvolvimento da abordagem utilizando regras de produção;
- Desenvolvimento da abordagem utilizando o método de força bruta, no qual o algoritmo testa todos os casos possíveis;
- Desenvolvimento da abordagem utilizando algoritmo genético;
- Realização do estudo empírico dos métodos para avaliar a acurácia dos mesmos;
- Comparação e análise dos métodos criados, mostrando vantagens e desvantagens de cada método.

1.3 Contribuições e Resultados

Esse trabalho apresenta três métodos para encapsular a complexidade de definir nós ranqueados, aumentando sua aplicabilidade. Os métodos foram desenvolvidos apresentando diferentes abordagens de acordo com seus objetivos específicos.

O primeiro método, publicado em Silva *et al.* [41], é baseado em regras de produção e compatível com o *AgenaRisk* [1]. Para definir as regras, dados foram coletados de um especialista.

O segundo foi desenvolvido em colaboração com um aluno do Instituto Federal da Paraíba. O mesmo é compatível com um algoritmo desenvolvido pelo aluno em questão para implementar nós ranqueados com suporte para WMEAN. Nesta pesquisa, complementou-se este algoritmo para ter suporte para WMIN, WMAX e MIXMINMAX. Este método utiliza a força bruta para definir a melhor configuração das TPC de acordo com dados elicitados de um especialista de domínio. Ou seja, o processo é demorado e custoso e deve ser usado caso o usuário não considere a variável tempo relevante.

O terceiro é uma melhoria do segundo com o objetivo de melhorar o desempenho. Para tal, foi utilizado algoritmo genético usando o *Brier score* como função-objetivo. O tempo

foi reduzido consideravelmente com relação ao método de força bruta, porém existe a possibilidade de não retornar a melhor combinação.

Para validar os métodos, utilizou-se a Rede Bayesiana apresentada em Perkusich *et al.* [35]. A mesma tem como objetivo detectar problemas em processos de software. Para tal, a acurácia das TPC definidas com os métodos propostos nesta pesquisa foi comparada com as de Perkusich *et al.* [35].

1.4 Estrutura da Dissertação

O restante deste documento está organizado nos capítulos descritos a seguir.

- No Capítulo 2, a fundamentação teórica da pesquisa é apresentada. Nós Ranqueados, Distribuição Nornal Truncada, Algoritmo Genético, Regras de Produção e Redes Bayesianas são introduzidos.
- No Capítulo 3, os métodos propostos são apresentados, assim como uma avaliação de cada um deles, destacando suas vantagens e desvantagens, cuidados necessários para aplicá-los e limitações.
- No Capítulo 4, são discutidas algumas pesquisas relacionadas com este trabalho, suas limitações e as contribuições desta pesquisa no intuito de melhorá-las;
- Por fim, as considerações finais, limitações do trabalho e propostas de trabalhos futuros são apresentadas no Capítulo 5.

Capítulo 2

Fundamentação Teórica

2.1 Redes Bayesianas

Redes Bayesianas são modelos de representação do conhecimento que trabalham com o fato deste ser incerto e incompleto por meio do Teorema de Bayes. Trata-se de um Grafo Acíclico Dirigido (GAD) onde os nós representam variáveis aleatórias e os arcos representam a dependência condicional entre as variáveis, respectivamente, são representados por círculos (ou elipses) e setas. Rede Bayesiana pertence à família de modelos gráficos probabilísticos e é usada para representar incertezas de um domínio [4]. A Figura 2.1 ilustra um exemplo de Rede Bayesiana.

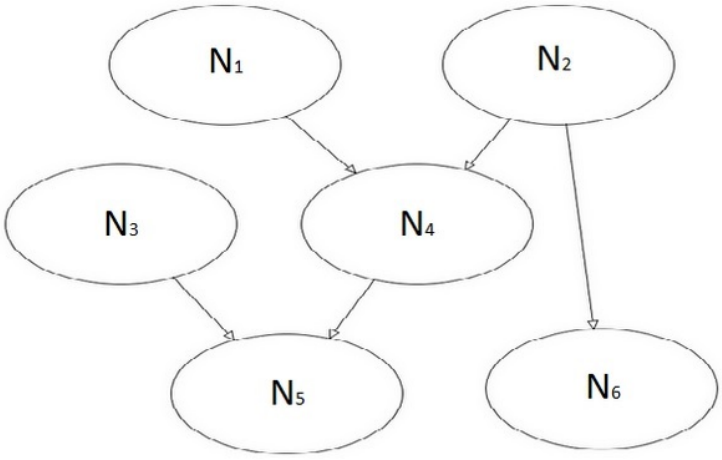


Figura 2.1: Rede Bayesiana

Além do grafo acícliclo direcionado, Redes Bayesianas definem as condições de distri-

buição de probabilidade para cada nó. As condições de distribuição de probabilidade são definidas em função apenas dos *pais* e são, geralmente, representadas por tabelas.

Um arco do nó N_i ao nó N_j representa a dependência condicional entre essas variáveis. O valor da variável N_j depende do valor da variável N_i. Dessa forma, N_i é chamado de *pai* e N_j de *filho*. Assim, o conjunto de *descendentes* é definido como o conjunto de nós que podem ser alcançados diretamente pelo nó. O conjunto de *ancestrais* é definido como os nós que podem alcançar o nó diretamente. Nota-se que apesar dos arcos representarem a direção da conexão causal entre as variáveis, informações podem propagar em qualquer direção no grafo [22].

Formalmente, uma Rede Bayesiana B é um GAD que representa a distribuição conjunta de probabilidade sobre o conjunto V de variáveis aleatórias [16]. A rede é definida pelo par $B = \{G, \Theta\}$. G é o GAD cujo os nós $N_1, \ldots, N_n$ representam variáveis aleatórias, e os arcos representam dependências diretas entre essas variáveis. O grafo G codifica suposições de independência, nas quais cada variável N_i é independente dos seus *não-dependentes* dados seus *pais* em G. Θ representa o conjunto de parâmetros da rede. Esse conjunto contém o parâmetro $\theta_{n_i|\pi_i} = P_B(n_i|\pi_i)$ para cada n_i em N_i condicionado por π_i, o conjunto de parâmetros de N_i em G. A Equação 2.1 apresenta a distribuição conjunta definida por B sobre V.

$$P_B(N_1, \ldots, N_n) = \prod_{i=1}^{n} P_B(n_i|\pi_i) = \prod_{i=1}^{n} \theta_{N_i|\pi i} \tag{2.1}$$

As funções de probabilidade podem ser representadas por tabelas-verdade. Na Figura 2.2, ilustra-se um exemplo fictício trivial de uma Rede Bayesiana. Nota-se que o valor do nó Câncer de Pulmão depende do valor dos seus pais. Apesar das arestas das Redes Bayesianas representarem o relacionamento causal entre as variáveis, a informação pode propagar em qualquer direção no grafo [3]. Ou seja, é possível calcular as probabilidades de um nó pai se apenas o valor do seu filho for conhecido. Por exemplo, na Figura 2.2, caso uma pessoa tenha parentes com câncer e seja fumante, é provável que essa pessoa desenvolva câncer de pulmão.

Redes Bayesianas possuem diversas características positivas, como facilidade de lidar com bases de dados pequenas e/ou incompletas, possibilidade de aprendizado, combinação de diferentes fontes de conhecimento, tratamento explícito de incertezas e rápido tempo de

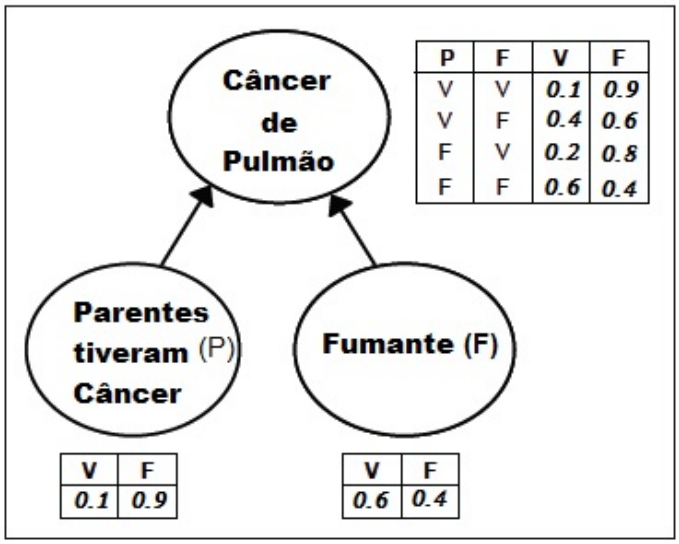


Figura 2.2: Rede Bayesiana com Tabelas de Probabilidade

respostas [46]. Por esse motivo, elas são utilizadas em sistemas que possuem algum tipo de incerteza envolvida [14]. O uso de Redes Bayesianas pode ser observado em sistemas especialistas, por exemplo, para auxiliar a tomada de decisões seguras em ambientes de projetos complexos [49] ou, ainda, para predizer o desempenho em projetos de inovação que adotam liderança transformacional [9].

2.2 Nós Ranqueados (Ranked Nodes)

Em Fenton *et al.* [31], no conceito de nó ranqueado, variáveis discretas cujos estados são expressos em uma escala ordinal (e.g., Baixo, Moderado e Alto) modeladas em uma escala contínua ordenada monotonamente no intervalo [0,1] foram apresentadas. Um exemplo que demonstra a modelagem de um nó ranqueado é apresentado na Figura 2.3.

Nós ranqueados tornam-se úteis na definição de Redes Bayesianas de larga escala quando não há dados suficientes para a utilização de *batch learning* [18]. Cabe ao usuário definir quantos estados cada nó terá, por outro lado, recomenda-se utilizar cinco ou sete estados [39]. Desta forma, nesta pesquisa, será utilizada a escala de cinco estados como base para calibração da TPN.

De acordo com Fenton *et al.* [31], o crucial sobre nós ranqueados é que eles podem fazer o trabalho de construção e edição em redes Bayesianas muito mais simples do que é possível

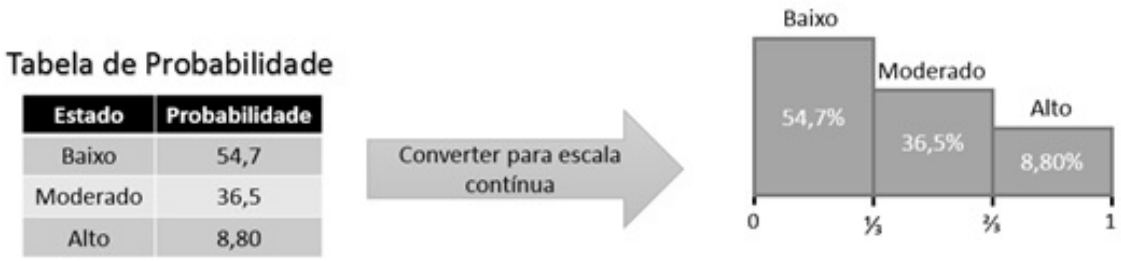


Figura 2.3: Conversão da Tabela de Probabilidade

quando os nós são subjetivos, uma vez que a Tabela de Probabilidade dos Nós (TPNs) requer um especialista para análise, o que é raramente rentável para obter conjuntos completos de valores de probabilidade. Em particular, desde que aparecem nas combinações adequadas, a complexidade da tarefa de construir TPN associadas é drasticamente simplificada.

2.3 Distribuição Normal Truncada (TNormal)

Após a conversão das TPNs para escala contínua gerando amostras que representam distribuições de probabilidade, o próximo passo é misturar as distribuições (i.e., amostras). Mistura-se as amostras que representam as TPNs dos nós pais com o objetivo de gerar uma distribuição normal truncada (TNormal).

Segundo Fenton *et al.* [31], a vantagem de utilizar a distribuição TNormal é que a mesma pode ter diversos formatos dependendo de sua média e variância. A média (μ) representa a tendência central da distribuição normal. A variância (σ^2), por sua vez, define a confiança no resultado e, consequentemente, a curva (i.e., quanto maior a variância, maior a tendência da curva se assemelhar com uma distribuição uniforme). Esta relação é demonstrada na Figura 2.4.

Precisa-se definir quatro parâmetros para gerar uma TNormal: a média da distribuição normal (μ), a variância da distribuição normal (σ^2), os limites inferiores e superiores. Dado que a TPN é limitada entre zero (0) e um (1), a distribuição TNormal também é. Desta forma, apenas necessita-se definir μ e σ^2.

Para definir a μ, utiliza-se as amostras definidas para representar as TPN dos nós pais e seus respectivos pesos. O peso define a influência do nó pai no nó filho e é definido por uma

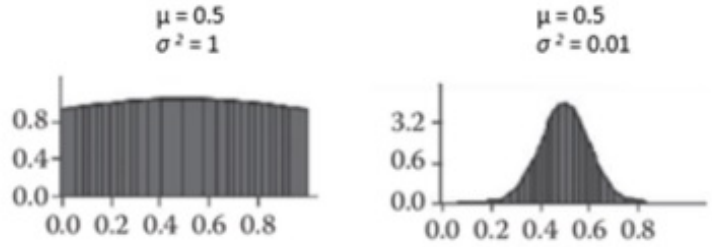


Figura 2.4: Exemplo de distribuição TNormal com a mesma μ, mas σ diferentes

constante *w* na qual $w \in N$. O mesmo é identificado a partir da equação ponderada utilizada para definir a TPN do nó filho. Por exemplo, para o nó C cuja equação é $C = 2A + B$, o peso do nó A é 2 e do nó B é 1. μ é um vetor com o mesmo número de elementos dos nós pais (i.e., 100000). Cada elemento de μ é calculado a partir de (2.2), na qual w_i é o peso, x_i é um elemento da amostra em questão e *n* é o número de pais.

$$\mu_j = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i} \tag{2.2}$$

Após definir μ, necessita-se definir σ^2 a partir da curva desejada (i.e., a confiança nos resultados). Ou seja, σ^2 é um valor manualmente definido pelo usuário. Finalmente, a fase de "misturar amostras" é concluída ao gerar uma TNormal com os parâmetros definidos.

Por fim, para definir a TPN do nó filho, a TNormal deve ser convertida em uma escala ordinal. Para tal, divide-se a densidade da TNormal nos pontos de interesse (i.e., entre 0 e 1/3, 1/3 e 2/3, e 2/3 e 1). Um exemplo de tal é apresentada na Figura 2.5.

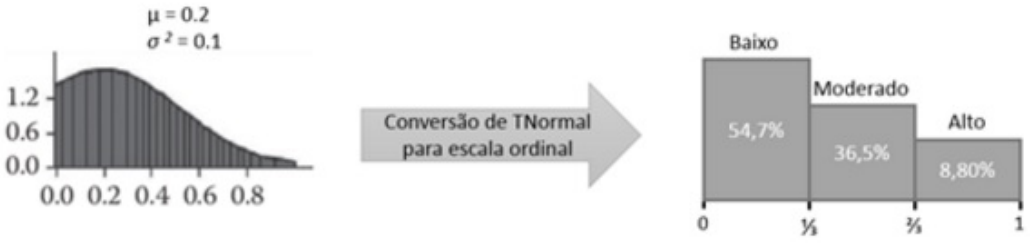


Figura 2.5: Exemplo de conversão de TNormal para escala ordinal

2.3.1 Funções de Probabilidade Condicional

Depois da definição de como os estados dos nós ranqueados serão mapeados para uma escala normalizada, é preciso selecionar uma expressão que melhor descreve a relação probabilística entre os nós pais e o nó filho. Tal expressão agrega pesos ponderados de escalas normalizadas dos nós pais para um ponto na escala normalizada do nó filho.

Segundo Laitila *et al.* [26], na prática, funções de probabilidade condicionais são funções que determinam a tendência central do nó filho dada uma combinação de estados dos nós pais. Existem quatro funções apresentadas em Fenton *et al.* [31]:

1. Média ponderada (WMEAN);
2. Mínimo ponderado (WMIN);
3. Máximo ponderado (WMAX);
4. Mistura de mínimo e máximo (MIXMINMAX).

As características das funções de probabilidade condicionais são descritas utilizando o exemplo da RB referenciada na Figura 2.6.

Caso a expressão escolhida seja WMEAN, WMIN ou WMAX, o peso w_i precisa ser atribuido a cada nó pai. Caso a expressão usada seja MIXMINMAX, a tendência central do nó filho é determinada somente pelo melhor e pior estado na combinação de estados dos nós pais. Dessa forma, apenas dois pesos, denotados por w_{min} e w_{max}, precisam ser atribuídos nesse caso. Para todas as funções, o peso que se aplica, quanto maior for o peso, mais forte é a influência do nó pai correspondente.

2.3.2 WMEAN

Na WMEAN, espera-se que o nó filho no intervalo [0,1] corresponda ao peso médio dos estados dos nós pais [0,1]. A tendência central é característica dessa função, com relação aos nós pais, porém, essa tendência pode mudar caso haja uma influência diferenciada dos nós pais com relação ao nó filho.

Como Laitila *et al.* [26] descrevem, a distribuição de probabilidade na Tabela de Probabilidade Condicional é gerada pela repetição da mesma rotina do cálculo para cada combinação de estados dos nós pais. Este ciclo de rotina é descrito a seguir.

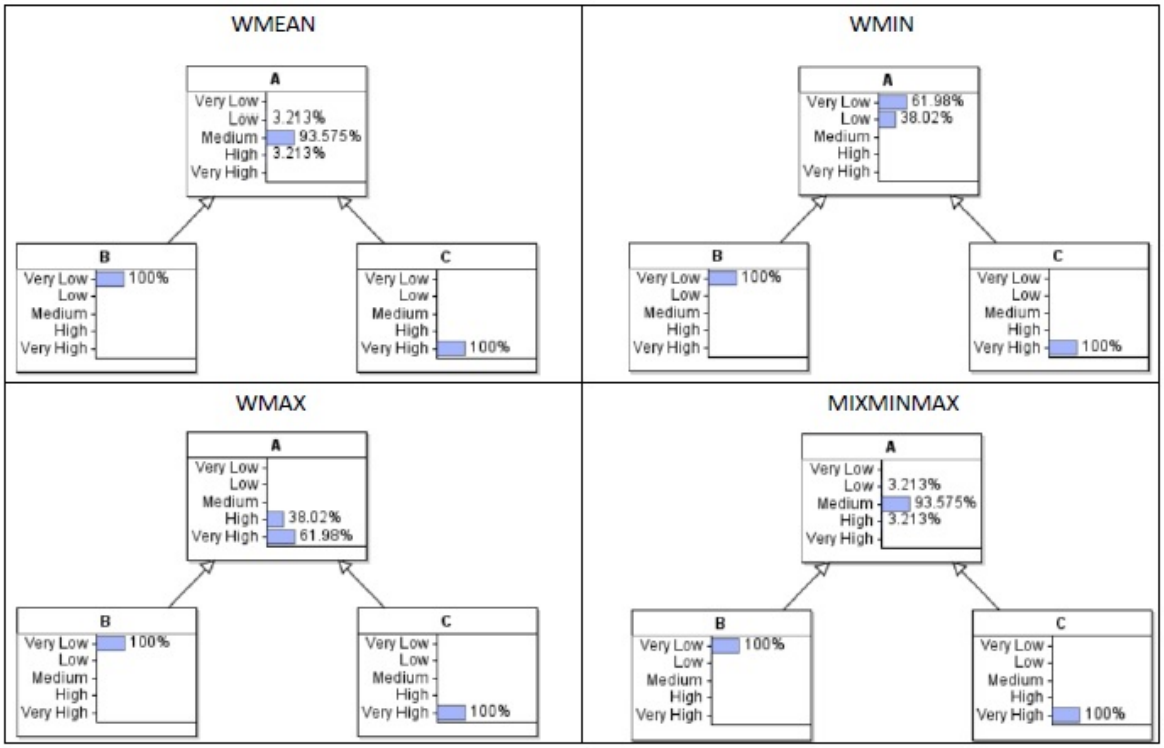


Figura 2.6: Exemplo da influência das Funções de Probabilidade Condicionais

Seja $(x_1; ...; x_n)$ uma dada combinação de estados dos nós pais e considere que $(z_1; ...; z_n)$ denotam os intervalos de estado identificados com eles, respectivamente. Uma combinação de estados $(x_1; x_2; x_3) = (Baixo, Alto, Baixo)$ do nós pais, pode ser identificada como uma combinação (z_1, z_2, z_3) = ([0; 1/3); [2/3; 1]; [2/3; 1]) dos intervalos de estado.

Depois de determinar os intervalos de estado $(z_1; ...; z_n)$, pontos de amostragem equidistantes $\{z_i, j\}_s^{j=1}$ são tomados a partir de cada um deles. Isto é realizado de modo que o primeiro ponto de amostra é o limite inferior do intervalo dado e o último é o limite superior. Assim, os pontos de amostragem são definidos de forma determinística. A partir de $n * s$ pontos de amostra retirados, s^n combinações de pontos de amostra $\{(z_1, k, ..., z_n, k\}_{s^n}^{k=1}$ são formadas de modo que, em cada combinação, há um ponto de amostragem a partir de cada nó pai. Por exemplo, utilizando-se $s = 5$, os pontos de amostra retirada de z_1 determinadas acima são $\{z_1, j\}_{j=1}^5 = \{0, 0.0833, 0.1667, 0.25, 0.333\}$. Para z_2 e z_3, os pontos de amostragem são $\{z_2, j\}_{j=1}^5 = \{z_3, j\}_{j=1}^5 = \{0.6667, 0.75, 0.8333, 0.3167, 1\}$. A partir desses pontos de amostragem, $5^3 = 125$ combinações $\{(z_1, p, z_2, r, z_3, q)\}_{p,r,q=1}^5$ são então formados. Estas incluem combinações tais como $(z_1, 1, z_2, 1, z_3, 1) = (0, 0.6667, 0.6667)$ e

$(z_1, 3, z_2, 1, z_3, 5) = (0.1667, 0.6667, 1)$.

Quando as combinações dos exemplos $\{z_1, k, ..., z_n, k\}_{k=1}^{s^n}$ são formados, a expressão de probabilidade escolhida é usada para calcular o valor médio correspondente $\{\mu_k\}_{s^n}^{k=1}$ para todos eles.

Nesse caso, a média μ é calculada pela Equação 2.3.

$$\mu_k = WMEAN(z_1, k, ..., z_n, k, w_1, ..., w_n) = \frac{\sum_{i=1}^{n} w_i * z_i, k}{\sum_{i=1}^{n} w_i} \tag{2.3}$$

Onde z_i, k é o ponto da amostra do $i$ésimo nó pai na $k$ésima combinação dos pontos da amostra e w_i é o peso do $i$ésimo nó pai.

2.3.3 WMIN

Na WMIN, o nó filho tende a seguir o estado do nó pai que tem o menor estado na escala normalizada [0,1]. A intensidade dessa tendência depende da influência dos nós pais no nó filho.

Nesse caso, a média μ é calculada pela Equação 2.4.

$$\mu_k = WMIN(z_1, k, ..., z_n, k, w_1, ..., w_n) = min_{i=1,...,n}\{\frac{w_i * z_i, k + \sum_{j \neq i}^{n} z_j, k}{w_i + n - 1}\} \tag{2.4}$$

2.3.4 WMAX

A WMAX tem a mesma função da WMIN, só que em direção contrária, ou seja, o nó filho tende a seguir o estado do nó pai que tem maior estado na escala normalizada [0,1]. Da mesma forma que a WMIN, a intensidade da tendência da escala do nó filho depende da influência dos nós pais.

Nesse caso, a média μ é calculada pela Equação 2.5.

$$\mu_k = WMIN(z_1, k, ..., z_n, k, w_1, ..., w_n) = max_{i=1,...,n}\{\frac{w_i * z_i, k + \sum_{j \neq i}^{n} z_j, k}{w_i + n - 1}\} \tag{2.5}$$

A Figura 2.7 ilustra a média μ para (a)WMEAN, (b)WMIN e (c)WMAX, com a influência dos pesos, dado que $w_1 = 5$ e $w_2 = 2$.

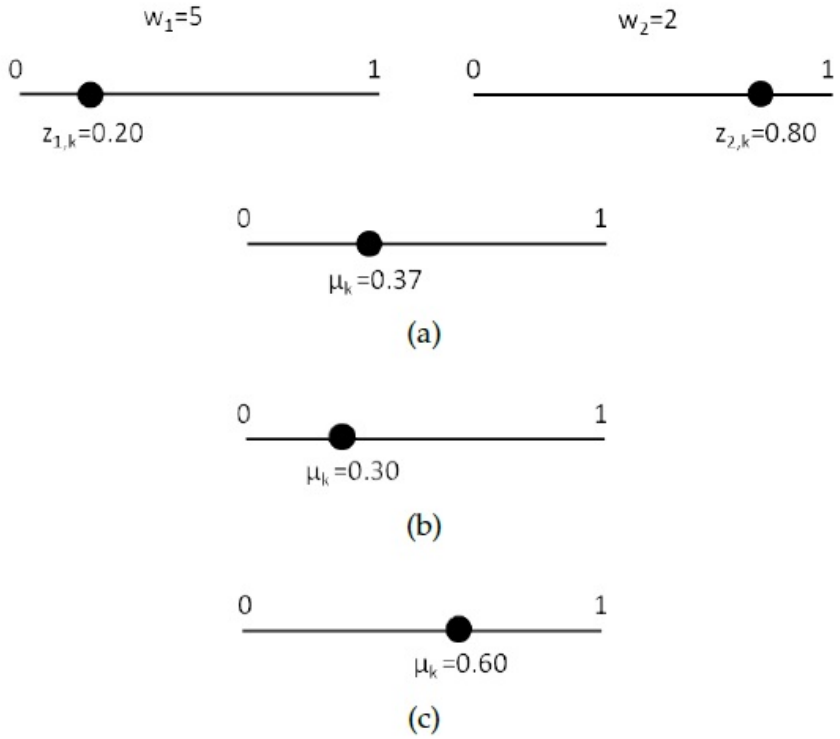


Figura 2.7: Resultado da μ_k utilizando pontos de amostra de nós pais e funções de probabilidade (a)WMEAN, (b)WMIN, c(WMAX)

2.3.5 MIXMINMAX

Na MIXMINMAX, a tendência central do nó filho é determinada pelo peso médio do maior e menor estado na escala normalizada encontrada da combinação dos estados dos nós pais.

Nesse caso, a média μ é calculada pela seguinte fórmula:

$$\mu_k = MIXMINMAX(z_1, k, ..., z_n, k, w_{min}, ..., w_{max}) = \frac{w_{min} * min_{i=1,...,n}\{z_i, k\} + w_{max} * max_{i=1,...,n}\{z_i, k\}}{w_{min} + w_{max}} \tag{2.6}$$

2.4 Algoritmo Genético

Um algoritmo genético é uma técnica utilizada na ciência da computação para achar soluções aproximadas em problemas de otimização e busca. Nos algoritmos genéticos, inicialmente é gerada uma população formada por um conjunto aleatório de indivíduos que podem ser vistos como possíveis soluções do problema. A evolução geralmente se inicia a partir de um

conjunto de soluções criado aleatoriamente e é realizada por meio de gerações. A cada geração, a adaptação de cada solução na população é avaliada, alguns indivíduos para a próxima geração, e recombinados ou mutados para formar uma nova população. Os membros mantidos pela seleção podem sofrer modificações em suas características fundamentais através de mutações e cruzamento (*crossover*) ou recombinação genética gerando descendentes para a próxima geração. Este processo, chamado de reprodução, é repetido até que uma solução satisfatória seja encontrada.

Os Algoritmos Genéticos diferem dos métodos tradicionais de busca e otimização, principalmente nos seguintes aspectos:

- Trabalham com uma codificação do conjunto de parâmetros e não com os próprios parâmetros;
- Trabalham com uma população e não com um único ponto;
- Utilizam informações de custo ou recompensa e não derivadas ou outro conhecimento auxiliar;
- Utilizam regras de transição probabilísticas e não determinísticas.

Algoritmos Genéticos são muito eficientes para busca de soluções ótimas, ou aproximadamente ótimas em uma grande variedade de problemas, pois não impõem muitas das limitações encontradas nos métodos de busca tradicionais. Além de ser uma estratégia de gerar-e-testar formal, por serem baseados na evolução biológica, são capazes de identificar e explorar fatores ambientais e convergir para soluções ótimas, ou aproximadamente ótimas em níveis globais.

A ideia básica de funcionamento dos algoritmos genéticos é a de tratar as possíveis soluções do problema como “indivíduos” de uma “população”, que irá “evoluir” a cada iteração ou “geração”. Para isso é necessário construir um modelo de evolução onde os indivíduos sejam soluções de um problema. A execução do algoritmo pode ser resumida nos seguintes passos:

- Inicialmente escolhe-se uma população inicial, normalmente formada por indivíduos criados aleatoriamente;

- Avalia-se toda a população de indivíduos segundo algum critério, determinado por uma função que avalia a qualidade do indivíduo (função de aptidão ou "*fitness*");
- Em seguida, através do operador de "seleção", escolhem-se os indivíduos de melhor valor (dado pela função de aptidão) como base para a criação de um novo conjunto de possíveis soluções, chamado de nova "geração";
- Esta nova geração é obtida aplicando-se sobre os indivíduos selecionados operações que misturem suas características (chamadas "genes"), através dos operadores de "cruzamento" (*crossover*) e "mutação";
- Estes passos são repetidos até que uma solução aceitável seja encontrada, até que o número predeterminado de passos seja atingido ou até que o algoritmo não consiga mais melhorar a solução já encontrada.

Os principais componentes mostrados na Figura 2.8 são descritos a seguir em mais detalhes.

2.4.1 População

É o conjunto de indivíduos que estão sendo cogitados como solução e que serão usados para criar o novo conjunto de indivíduos para análise. O tamanho da população pode afetar o desempenho global e a eficiência dos algoritmos genéticos. Populações muito pequenas têm grandes chances de perder a diversidade necessária para convergir a uma boa solução, pois fornecem uma pequena cobertura do espaço de busca do problema. Entretanto, se a população tiver muitos indivíduos, o algoritmo poderá perder grande parte de sua eficiência pela demora em avaliar a função de aptidão de todo o conjunto a cada iteração, além de ser necessário o uso de melhores recursos computacionais.

2.4.2 Avaliação de Aptidão

A avaliação de aptidão (do inglês, *fitness function*) calcula, através de uma determinada função, o valor de aptidão de cada indivíduo da população. Este é o componente mais importante de qualquer algoritmo genético. É através desta função que se mede quão próximo um indivíduo está da solução desejada ou quão boa é esta solução.

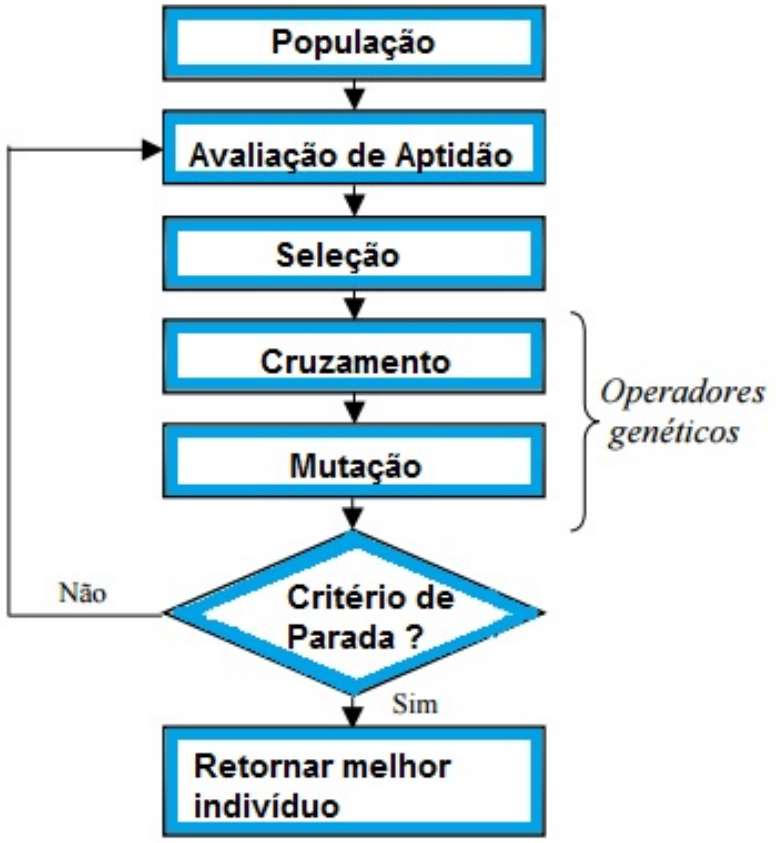


Figura 2.8: Estrutura básica de um Algoritmo Genético

É essencial que esta função seja muito representativa e diferencie na proporção correta as más soluções das boas. Se houver pouca precisão na avaliação, uma ótima solução pode ser descartada durante a execução do algoritmo, além de gastar mais tempo explorando soluções pouco promissoras.

2.4.3 Seleção

Dada uma população em que a cada indivíduo foi atribuído um valor de aptidão, aplica-se um dos vários métodos para selecionar os indivíduos sobre os quais serão aplicados os operadores genéticos.

2.4.4 Operadores Genéticos

O principio básico dos operadores genéticos é transformar a população através de sucessivas gerações, estendendo a busca até chegar a um resultado satisfatório. Os operadores genéticos são necessários para que a população se diversifique e mantenha características de adaptação adquiridas pelas gerações anteriores. Os operadores de cruzamento e de mutação têm um papel fundamental em um algoritmo genético.

2.4.5 Cruzamento

O operador de cruzamento (do inglês, *crossover*) é considerado o operador genético predominante. Através do cruzamento são criados novos indivíduos misturando características de dois indivíduos "pais". Esta mistura é feita de forma análoga (em um alto nível de abstração) a reprodução de genes em células biológicas. Trechos das características de um indivíduo são trocados pelo trecho equivalente do outro.

O resultado desta operação é um indivíduo que potencialmente combine as melhores características dos indivíduos usados como base. Alguns tipos de cruzamento bastante utilizados são o cruzamento em um ponto e o cruzamento em dois pontos, ilustrados nas Figuras 2.9 e 2.10.

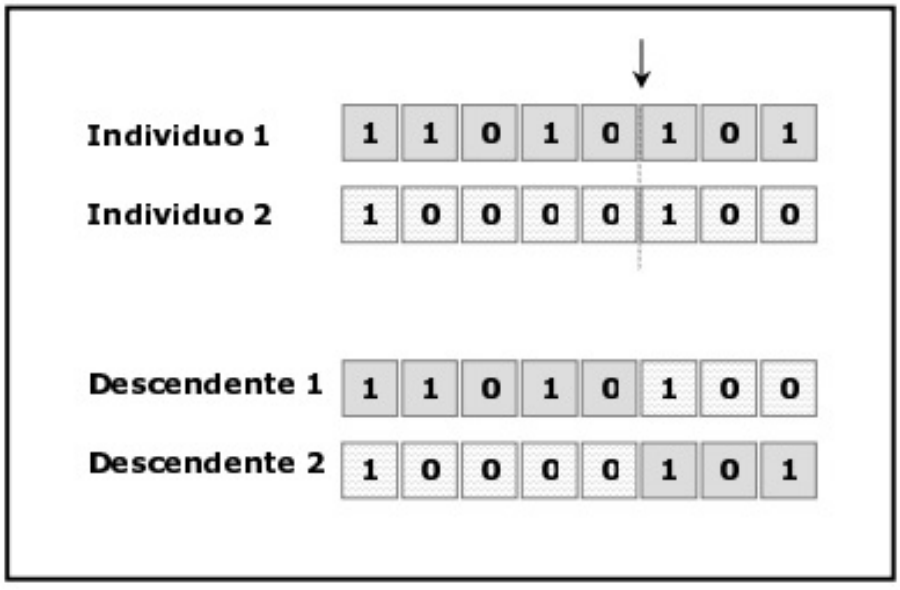


Figura 2.9: Cruzamento em um ponto

Com um ponto de cruzamento, seleciona-se aleatoriamente um ponto de corte do cromossomo, onde cada cromossomo representa uma solução. Cada um dos dois descendentes recebe informação genética de cada um dos pais (Figura 2.9).

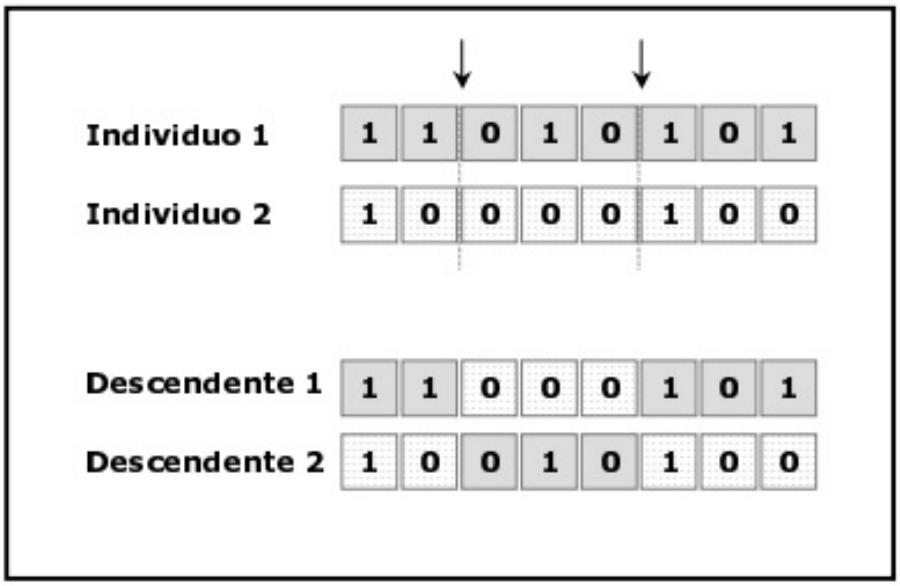


Figura 2.10: Cruzamento em dois pontos

Com dois pontos de cruzamento, um dos descendentes fica com a parte central de um dos pais e as partes extremas do outro pai e vice-versa (Figura 2.10).

2.4.6 Mutação

Esta operação simplesmente modifica aleatoriamente alguma característica do indivíduo sobre o qual é aplicada (ver Figura 2.11). Esta troca é importante, pois acaba por criar novos valores de características que não existiam ou apareciam em pequena quantidade na população em análise. O operador de mutação é necessário para a introdução e a manutenção da diversidade genética da população. Desta forma, a mutação assegura que a probabilidade de se chegar a qualquer ponto do espaço de busca possivelmente não será zero. O operador de mutação é aplicado aos indivíduos através de uma taxa de mutação geralmente pequena.

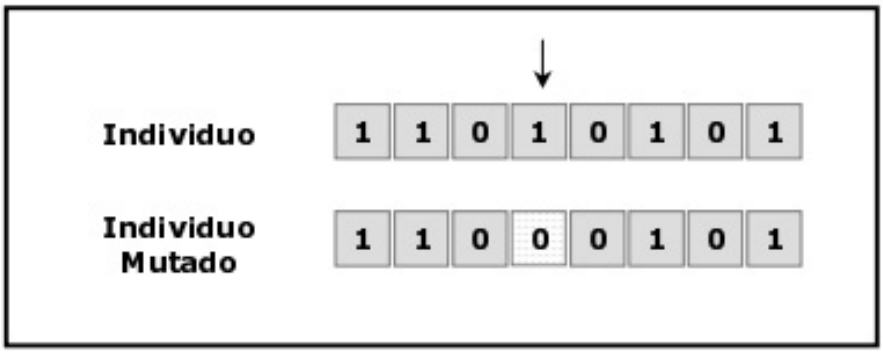


Figura 2.11: Mutação Simples

2.4.7 Geração

Dá-se o nome de "Geração" a um novo conjunto de indivíduos que é gerado a partir da população anterior, a cada passo gerado. É através da criação de uma grande quantidade de gerações que é possível obter melhores resultados dos Algoritmos Genéticos.

2.5 Regra de Produção

Na Inteligência Artificial simbólica, a representação do conhecimento é realizada por meio de uma coleção de símbolos e com procedimentos de interpretação. Existem distintos tipos de representação, entre eles está a Regra de Produção (do inglês, *Production Rules*).

As regras de produções são a técnica de representação do conhecimento mais usada, e isto se deve a principalmente a sua facilidade de compreensão e programação [29]. É um sistema mais intuitivo que os demais, no entanto não apresenta uma semântica lógica clara, normalmente utilizando um encadeamento progressivo e permitindo um raciocínio não monótono.

Esse tipo de regra representa conhecimento com pares de condição-ação, SE condição (ou premissa ou antecedente) ocorre ENTÃO ação (resultado, conclusão ou consequente) deverá ocorrer. São denominadas regras de produção porque quando utilizadas com raciocínio progressivo, produzem novos fatos e regras de base de conhecimento já existente.

As regras de produção são formadas por pares de condição-ação e podem ser vistas como uma simulação do comportamento cognitivo de especialistas humanos, representando o conhecimento de forma modular, em que cada regra apresenta um "pedaço" de conhecimento

independente, no entanto permanece a consistência do mesmo [47].

Os sistemas baseados em regras de produção separam o conhecimento permanente do conhecimento temporário (base de regras e memória de trabalho), seus módulos são estruturalmente independentes, sua modularidade facilita a independência funcional, além de ser possível utilizar uma variedade de esquemas de controle [47].

Capítulo 3

Métodos Propostos

Foram elaborados três métodos com o objetivo de permitir a definição de TPC por especialistas sem conhecimento específico de nós ranqueados. O primeiro método [41] é baseado em regras de produção (*production rules*). Para criar as regras, foram elicitados dados a partir de um especialista em nós ranqueados. Para validar essa nova abordagem, realizou-se um experimento com uma Rede Bayesiana já publicada na literatura para verificar se, com a nova abordagem, um usuário pode conseguir a mesma ou uma melhor configuração para a TPC. Esta solução é compatível com a implementação de nós ranqueados do AgenaRisk [1]. Mais detalhes do método na Seção 3.1.

Dado que o AgenaRisk [1] é uma ferramenta paga e limitada, o que restringe as possibilidades de utilização de nós ranqueados, um algoritmo independente para nós ranqueados foi desenvolvido em colaboração com um aluno do Instituto Federal da Paraíba (IFPB) - Campus Monteiro. Esse algoritmo, publicado na literatura [5] utiliza algoritmos de baixo nível (e.g., geração de amostras de distribuição Normal truncada) diferentes do AgenaRisk. Desta forma, a primeira solução não pode ser reutilizada.

Para tal, foram definidas duas soluções. Em ambas, foi utilizado o *Brier score* para avaliar a precisão da TPC e avaliar os resultados com o teste de Wilcoxon. A diferença é o método de escolha da melhor combinação. O método utiliza uma abordagem de força bruta. Ou seja, no caso de Redes Bayesianas de larga escala, o processamento é custoso. Por outro lado, pode-se encontrar o melhor resultado possível de acordo com os dados de entrada. Esta solução é apresentada em mais detalhes na Seção 3.2.

Com o intuito de ter uma solução de melhor desempenho, o terceiro método é baseado em

uma abordagem de otimização evolucionária e utiliza algoritmos genéticos. Este utiliza um número fixo para a quantidade de combinações ou um número ótimo de *Brier Score* como condição de parada. Dessa forma, o processamento é mais ágil, porém, existe a possibilidade de ter que executar essa solução mais de uma vez para um melhor resultado. Esta solução é apresentada em mais detalhes na Seção 3.3.

3.1 Método Regras de Produção

3.1.1 Descrição

Publicado em Silva *et al.* [41], este método é baseado em regras de produção, é compatível com o *AgenaRisk* [1] e automatiza a abordagem apresentada em Fenton *et al.* [31]. Para criar as regras, foram elicitados dados de um especialista em redes Bayesianas para, a cada conjunto de combinações de evidências dos nós pais, definir a melhor configuração da TPC.

Mais especificamente, o objetivo é combinar a capacidade de modelagem da abordagem apresentada em Fenton *et al.* [31] e encapsular o conhecimento específico em nós ranqueados da abordagem apresentada em Perkusich *et al.* [34]. Para elicitar o conhecimento de especialistas, como em Fenton *et al.* [31], foi utilizada a análise "E SE" (isto é, resultados da tabela verdade). Dadas as informações coletadas, a calibração das funções de probabilidades condicionais foi automatizada.

Para calibrar a função de um nó ranqueado, é necessário definir três parâmetros: *f*, V = (v_1, ..., v_k) e (σ^2), onde *f* é o tipo de função, V é o vetor contendo os pesos dos nós pais e *k* é o número de nós pais. No *AgenaRisk* [1], essas variáveis possuem os seguintes alcances: $f \in \{WMEAN, WMIN, WMAX, MIXMINMAX\}$, $w \in \{1, ..., 5\}$, $\sigma^2 \in \{5.0\text{E}^{-}4$, ..., $\infty\}$ e $k \in \{1, ..., \infty\}$.

Dado que os parâmetros das funções de probabilidade condicional são definidos, é possivel avaliar a acurácia dos cálculos (isto é, previsões) com o *Brier score* . Para uma única previsão, que é o caso, é simplesmente o quadrado da diferença entre a probabilidade prevista (*q*) e o resultado os valores reais (*o*) [31], para cada estado: $B = \sum_{n=1}^{s}(o_n - q_n)^2$, onde *B* é o *Brier score* e *s* é o número de possíveis resultados (isto é, número de estados do nó dado). Dado que o objetivo é a melhor calibração possível, o problema é, tendo em vista os

dados coletados dos especialistas, encontrar uma combinação de parâmetros f e $V = (v_1, ..., v_n)$ que minimiza B.

Uma regra de produção consiste em duas partes: uma pré-condição sensorial (isto é, instrução SE) e uma ação (ou seja, ENTÃO). Se uma entrada para o sistema corresponde a uma pré-condição, uma ação é executada. Com regras de produção, é possível representar um conhecimento especializado. Por exemplo, a regra dada representa o nosso tráfego de conhecimento a respeito: "se o semáforo está vermelho depois parar". Para definir as regras, contou-se com o conhecimento de um especialista com cinco anos de experiência usando nós ranqueados. A solução foi implementada utilizando a ferramenta *Expert Sinta* [2].

Na ferramenta, o usuário visualiza a pergunta de acordo com a Figura 3.1. O *Expert Sinta* permite que o desenvolvedor escolha uma ou mais opções, porém, as regras foram desenvolvidas para que o usuário escolha apenas uma opção. É possível que o usuário diminua a confiança, como mostrado na Figura 3.2, que inicialmente é 100% quando uma opção é marcada. Essa confiança influencia na confiança da resposta.

Figura 3.1: Perguntas no *Expert Sinta*

O resultado é ilustrado juntamente com sua confiança na Figura 3.3. Algumas combinações são equivocadas por parte do usuário e uma configuração ideal não é encontrada, nesse caso, a ferramenta mostra essa informaçao ao usuário (Figura 3.4).

Em Fenton *et al.* [31], os autores apresentam o uso de uma tabela verdade composto por uma combinação de estados dos nós pais para coletar dados de especialistas do domínio. Portanto, o primeiro passo foi definir quais os valores da tabela verdade. Com a tabela verdade (isto é, combinações) definida, foi possível definir os pré-requisitos do sistema.

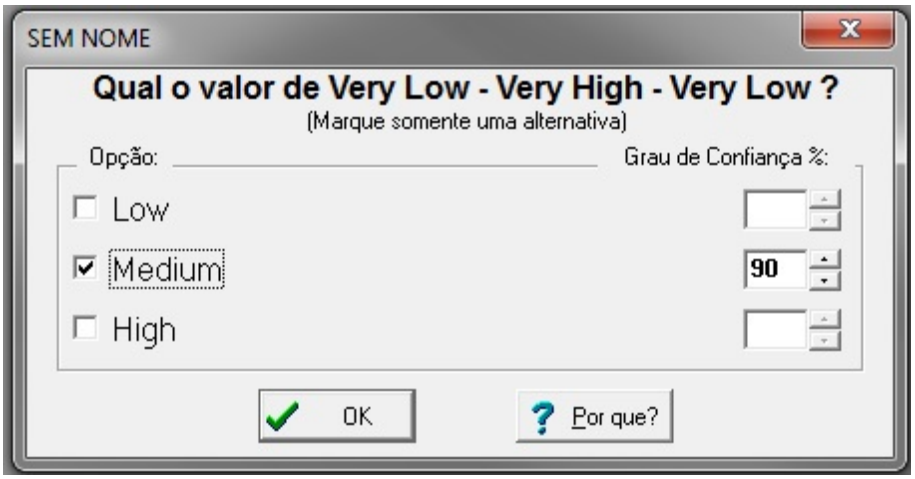


Figura 3.2: Exemplo de resposta com 90% de confiança

Para obter os pesos para WMAX e WMEAN, Laitila *et al.* [26] recomendam que o especialista especifique qual ponto (isto é, o estado) que o modo do nó filho aumenta quando $s_a = (0)_{i=1}^{n}$ mudar para $s_b = (0, ..., 0, s_k = 1, 0, ..., 0)$; e, para WMIN, o modo do nó filho cai quando $s_a = (1)_{i=1}^{n}$ muda para $s_b = (1, ..., 1, s_k = 0, 1, ..., 1)$. Assim, para cada configuração (isto é, o número de nós pai), consideram-se alguns casos, como mostrado no Apêndice A. As combinações utilizadas para calibrar um nó filho com três pais (A, B e C) possuem todos os nós compostos dos estados *s* = (*Muito Baixo, Baixo, Médio, Alto, Muito Alto*).

Então, o especialista define, para cada combinação possível na tabela de verdade, a melhor calibração para o EPC (isto é, a ação): tipo de função e pesos. Foi definido ($\sigma^2 = 5.0E^{-}4$) porque, de acordo com o especialista, alterar as funções e pesos é suficiente. De fato, em Perkusich [34], os autores definiram ($\sigma^2 = 5.0E^{-}4$) e obtiveram sucesso. Para os pesos, foi considerado o intervalo (1, ..., 5). Com valores mais altos, o resultado dificilmente muda de estado, o que muda geralmente é a porcentagem de confiabilidade na resposta, mas o resultado continua sendo o mesmo, como mostra o apêndice B. Por exemplo, se para a combinação (*Muito Alto; Muito Baixo*) o valor esperado é *Baixo*, para (*Muito Baixo; Muito Alto*), é *Baixo*, para (*Muito Baixo; Médio*) é *Muito Baixo* e para (*Muito Baixo; Muito Alto*) é *Muito Baixo*, então a melhor calibração é a função WMIN com pesos 3 e 5. Para verificar as calibrações, foi utilizado o *AgenaRisk* [1] e o Brier score. Para consolidar a regra, a média do Brier score para todas as combinções foi menor que 0.1.

Na Figura 3.5 são ilustradas as regras definidas pelo especialistas, no exemplo, um con-

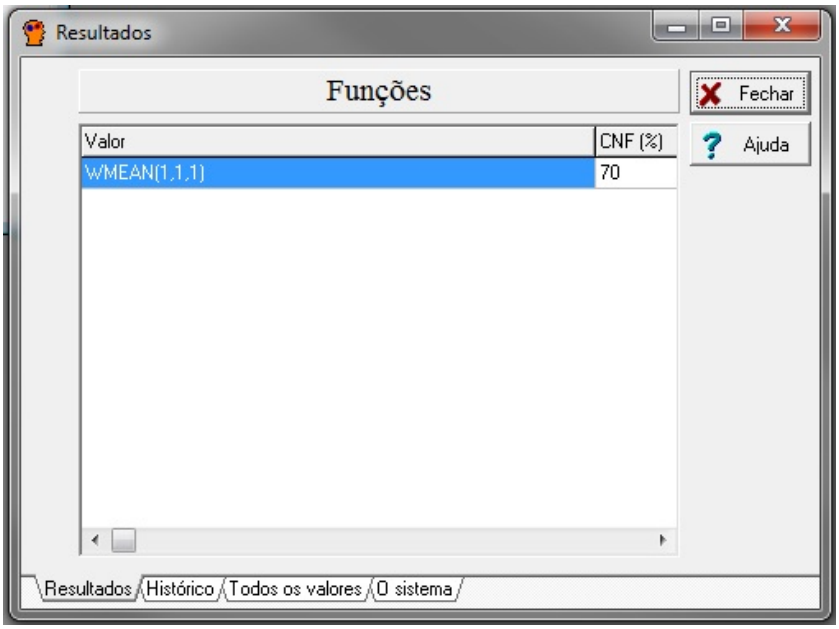


Figura 3.3: Exemplo de resultado do primeiro método no ExpertSinta

junto de combinações para uma amostra com três nós pais. As regras definidas para os nós filhos com dois e três nós pais, pois acima disso, uma separação dos nós é sugerida para simplificação da rede [31].

3.1.2 Avaliaçao

Para avaliar essa abordagem, foi executado um experimento utilizando uma rede Bayesiana já publicada na literatura como objeto de estudo. Foram selecionados aleatoriamente cinco nós da rede Bayesiana como os objetos de estudo para verificar se, com a abordagem de regras de produção, um praticante pode conseguir a mesma ou melhor configuração para a TPC. Para cada nó, foram selecionados aleatoriamente doze combinações de estados para extrair dados de um praticante, calculou-se o *Brier score* e avaliou os resultados com o teste de Wilcoxon. Todos os testes de Wilcoxon executados rejeitaram as hipóteses nulas que indicava que o *Brier score* para o método antigo era o mesmo que o novo.

Para avaliar a solução utilizando o método regras de produção, foi executado um experimento usando uma amostra de nós da rede Bayesiana como o objeto de estudo, figura 3.6, apresentada em Perkusich M., Gorgônio K. C., Almeida H., Perkusich A. *Assisting the Continuous Improvement of Scrum Projects using Metrics and Bayesian Networks.* Journal of Software: Evolution and Process. No prelo 2016 provável de publicação após aceite. A rede

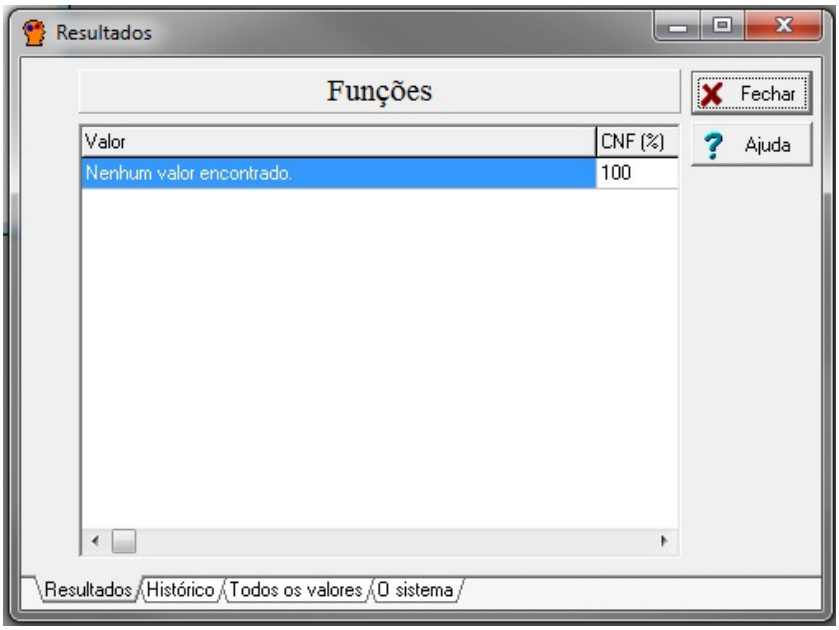


Figura 3.4: Exemplo de resultado não encontrado do primeiro método no ExpertSinta

A	B	C
Muito Alto	Muito Alto	Muito Baixo
Muito Alto	Muito Baixo	Muito Alto
Muito Baixo	Muito Alto	Muito Alto
Muito Baixo	Muito Baixo	Muito Alto
Muito Baixo	Muito Alto	Muito Baixo
Muito Alto	Muito Baixo	Muito Baixo

Figura 3.5: Tabela verdade para um nó filho com três pais

Bayesiana apresentada é uma melhoria da rede Bayesiana já publicada em Perkusich. *et al.* [34].A rede Bayesiana foi escolhida devido à disponibilidade e modelagem dos fatores-chave de projetos de software baseados em *Scrum* com o objetivo de auxiliar na melhoria contínua da equipe e processos. A rede Bayesiana é composta por vinte e um nós filho. Em outras palavras, existem vinte e uma TPC a ser calibrada. As TPC foram calibradas utilizando a abordagem apresentada em Perkusich *et al.* [34].

A avaliação consiste na aplicação do método regras de produção para calibrar uma amostra de TPC do objeto de estudo e comparar a precisão dos novas TPC com as antigas. Para este propósito, foi elicitado o conhecimento de um especialista que tem cinco anos de expe-

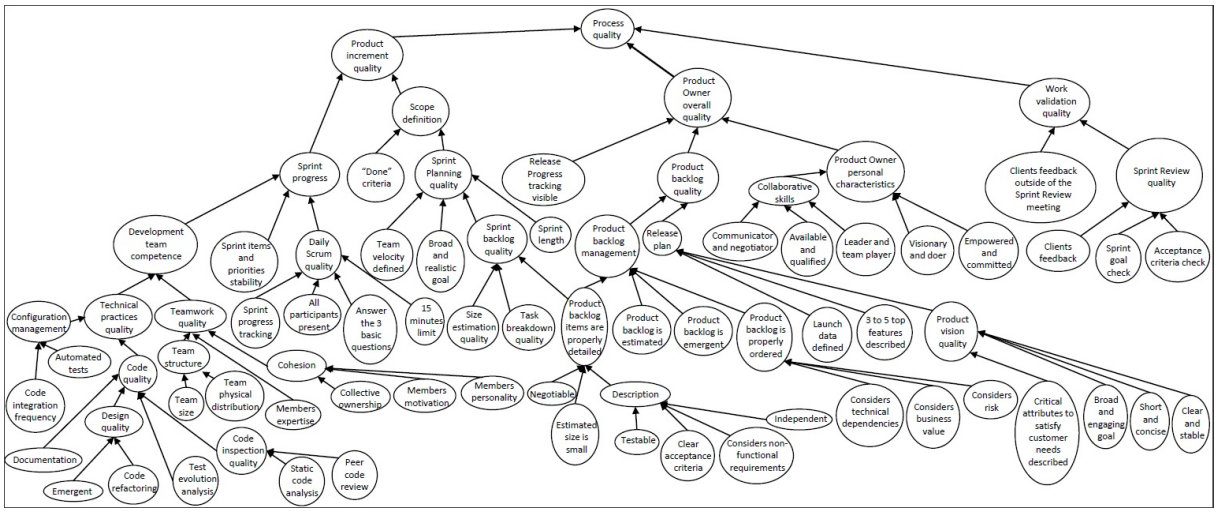


Figura 3.6: Rede Bayesiana redimensionada de Perkusich *et al.* [34]

riência trabalhando em projetos *Scrum* como *Scrum Master* e já estava familiarizado com a Rede Bayesiana apresentada em Perkusich *et al.* [28]. Primeiro, elicitou-se o conhecimento sobre o valor esperado da TPC para um conjunto de combinações dos estados dos nós pais, para usar como entrada para o método em questão, *I*. Depois, foi elicitado o conhecimento para um conjunto diferente, *E*, em que $E \in I = \emptyset$, e comparou-se com os valores calculados do sistema. Neste caso, uma vez que se compara os dados gerados pelo método regras de produção com os dados do sujeito, não existe conflito em ter o mesmo sujeito para calibrar e avaliar as TPC. Foca-se sobre a seguinte questão de pesquisa e hipótese nula informal:

RQ1: Comparado com o método antigo, usar a nova calibração mantem ou melhora a acurácia do modelo dado a expectativa do usuário?

H_0: A precisão diminui.

No Apêndice 2, é apresentado os dados coletados do *Scrum Master* para elicitar a TPC para todos os nós. Para o experimento, foram selecionados aleatoriamente cinco nós filhos, com dois ou três pais, da rede Bayesiana apresentada em Perkusich *et al.* [28] como os objetos de estudo. A variável resposta é a acurácia dos antigos e dos novos modelos, que são avaliados pelo *Brier score*. Para cada nó, foi definido aleatoriamente doze combinações de estados de estados dos nós pai e foi utilizado para, por meio da tabela de verdade, obter dados do especialista sobre a tendência central esperada do nó dado. Para cada combinação, foi calculado o *Brier score* usando na calibração apresentado em Perkusich *et al.* [28] e, usando a calibração do novo método. Foi utilizado a média do *Brier score* para comparar a

precisão dos modelos. Tendo em conta que os dados não seguem uma distribuição normal, foi utilizado o teste de Wilcoxon. Ao analisar os resultados dos testes de Wilcoxon, um para cada nó, avaliou-se **RQ1**.

Os objetos do estudo foram os nós: *Work validation quality, Product backlog quality, Software engineering techniques quality, Sprint Review quality* e *Product Backlog*, devidamente ordenados. A Figura 3.7, mostra os dados elicitados e *Brier score* calculados para o nó *Work validation quality*. Para o modelo antigo (isto é, Perkusich *et al.* [28]), a pontuação média do *Brier score* é 0.59. Para a novo método (isto é, Silva *et al.* [41]) o *Brier score* é 0.24. Aplicando o teste de Wilcoxon com $\alpha = 0.5$, tem-se o valor *p-value* = 0.0042. Portanto, rejeita-se a hipótese nula que afirma que a média do *Brier score* para as TPC definidas com a nova abordagem são piores do que o original.

Sprint Review quality	Clients feedback outside of the Sprint Review	Work validation quality	Old Brier score	New Brier score
Very low	High	Low	0.5	0.0085
Low	High	Low	1.8	0.97
High	Low	High	1.8	0.97
Low	Medium	Low	0.5	0.18
Medium	Low	Medium	0.5	0.18
Medium	Very high	High	0.0041	0.18
Very high	Medium	High	0.0041	0.18
Very high	Low	High	0.5	0.0077
High	Very low	Medium	0.5	0.0077
Low	Very high	Medium	0.5	0.0077
Medium	High	Medium	0.5	0.18
Very low	Very low	Very low	0.0021	0.0013

Figura 3.7: Coleta de dados e cálculo do *Brier score* para o nó *Work validation quality*

Para o nó *Product backlog quality*, obteve-se *p-valor* =0.0085. Para *Software engineering techniques quality*, *p-valor* = 0.0041. *Sprint Review quality p-valor* = 0.0033 e para *Product Backlog is properly ordered*, *p-valor* = 0.0017. Portanto, para todos os nós, concluiu-se que o novo modelo é mais preciso. Uma ameaça à validade é que seja possível não ter avaliado nós suficientes para avaliar **RQ1**.

3.1.3 Limitações

As limitações deste método estão relacionadas com a definição das regras de produção e ameaças à validade em relação ao experimento. Quanto à definição das regras de produção, só se baseou na experiência de um especialista para defini-los. Para minimizar esta limitação,

foi selecionado um especialista experiente em nós ranqueados e usado o *Brier score* para minimizar as chances de uma definição de regra incorreta. Além disso, o método proposto só lida nós filho com dois ou três pais. No entanto, na prática, esta não deve limitar a sua aplicação, porque sempre que um nó tenha mais de três pais, um refatoramento deve ser usado para simplificar a rede Bayesiana [31]. Além disso, as soluções foram definidas para os nós ranqueados compostos de uma escala de 5 pontos. Finalmente, a definição das regras baseiam-se em valores do AgenaRisk. Por outro lado, atualmente, é a única ferramenta que implementa os nós ranqueados.

Com relação às limitações do experimento, conclui-se que existem ameaças de conclusão, internas e externas à validade. As ameaças de conclusão à validade estão relacionadas com os tamanhos da amostra utilizadas para os objetos de estudo. A rede Bayesiana original foi composta de vinte e um nós filho e só foram avaliadas cinco. Além disso, para comparar as precisões, só foram avaliadas doze combinação de estados. As ameaças internas à validade estão relacionadas com o processo de seleção do assunto. Por outro lado, foi minimizado essa ameaça, escolhendo um especialista familiarizado com a rede Bayesiana, que minimizou a ameaça de elicitar conhecimento inconsistente. As ameaças externas à validade preocupam a capacidade de generalizar resultados da experiência fora do ambiente do experimento. Uma vez que apenas uma rede Bayesiana e um assunto, não se pode generalizar os resultados. No entanto, dado que os nós ranqueados são utilizados e os dados coletados a partir do especialista de domínio é consistente, não há nenhuma razão para acreditar que o novo método não iria apresentar dados precisos de saída.

3.2 Método Força Bruta

3.2.1 Descrição

Um aluno do Instituto Federal da Paraíba (IFPB), desenvolveu um algoritmo na linguagem de programação C++ [5], no qual foi implementado a distribuição normal e o cálculo da função de probabilidade WMEAN, simulando o método de Perkusich [34]. Esse algoritmo foi modificado com o objetivo de melhorar o método já existente, com isso, foi desevolvido o cálculo do *Brier Score* e as outras funções que não foram levadas em consideração

antes, WMIN, WMAX, MIXMINMAX. O algoritmo original recebia os parâmetros de entrada direto no código, esse método de entrada dos dados também foi alterado, para que o especialista insira os parâmetros necessários no código executável.

Assim como o método Regras de Produção, σ^2 também foi definido por $5.0E^{-}4$ e os pesos variando de 1 a 5, por motivos já explicados anteriormente. Os parâmetros necessários para o cálculo da função de probabilidade condicional mais adequada, que não foram definidos nos métodos, é a quantidade de nós pais para o nó filho em questão e os valores para cada combinação.

A primeira pergunta que o usuário precisa responder é a quantidade de nós pais do nó filho, como na Figura 3.8. De acordo com a resposta do especialista, as perguntas serão geradas. Para cada resposta, são gerados $2 * n$ perguntas, sendo n o número de nós pais. A resposta precisa ser em caixa alta e entre as combinações VL, L, M, H e VH que correspondem, respectivamente, à Muito Baixo, Baixo, Médio, Alto, Muito Alto. O método não possui tratamento de erro, ou seja, caso usuário não responda corretamente as questões, a aplicação é fechada.

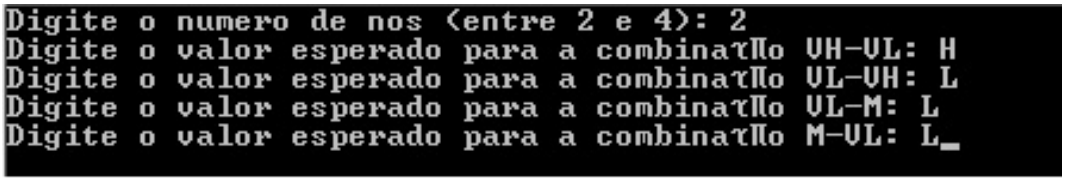


Figura 3.8: Método Força Bruta - Entrada de dados

A principal característica dessa abordagem, é que existe um laço no código que faz a combinação de todos os casos das quatro funções variando os pesos entre 1 e 5, no qual para cada combinação é calculado seu *Brier score*. Ao final do processamento, a combinação com o menor *Brier score* é considerado o melhor resultado.

O tempo de processamento varia com a quantidade de nós pais, quanto maior, maior o número de questões, maior o número de questões, e consequentemente, maior o tempo de processamento. Para um nó com dois pais, o tempo de processamento desse método é em torno de 6 minutos, para um nó com três pais, o tempo é em torno de 1 hora e 40 minutos e para um nó com quatro pais, esse tempo ultrapassa as 3 horas. Esse método tem a vantagem de testar todos os casos dentre as restrições apresentadas, porém, se o tempo de processamento for um fator importante, para nós com muitas entradas, talvez esse método

não seja o mais adequado.

3.2.2 Limitações

As limitações deste método estão relacionadas com o tempo de processamento e a disponibilidade de memória para execução do mesmo. O método testa todas as combinações variando os quatro tipos de funções e os pesos de cada nó pai entre um e cinco.

Quanto maior o número de nós pais, maior o número de combinações possíveis, consequentemente, maior é o tempo de processamento para testar cada combinação. Considerando o tempo uma variável fundamental na calibração da rede Bayesiana, tal método pode não ser o mais adequado para nós filhos com três ou quatro nós pais. O teste, para quatro nós pais, demorou mais de três horas e apresentou erro de memória na máquina em execução. Nesse caso, seria necessário aumentar a memória para tal processamento.

Com relação às limitações do experimento, conclui-se que existem ameaças de conclusão à validade. Estas estão relacionadas com quantidade de nós pais utilizadas para os objetos de estudo e memória disponível. A ameaça de erro de memória pode ser resolvida com uma máquina de melhor configuração, não foi possível testar para quatro nós com a máquina disponível, porém, um método que testa todos os casos dentro dos limites apresentados, garante maior confiança no resultado.

3.3 Método Genético

3.3.1 Descrição

Na busca de melhorar a eficiência do método Força Bruta, procurou-se otimizar o código utilizando algoritmo genético. Os algoritmos genéticos utilizam conceitos provenientes do princípio de seleção natural para abordar uma série ampla de problemas, em especial de otimização. Robustos, genéricos e facilmente adaptáveis, consistem de uma técnica amplamente estudada e utilizada em diversas áreas. É possível ver a estrutura básica do algoritmo na Figura 2.8.

Para a mutação, é escolhido aleatoriamente um peso ou função, uma vez que a variância é fixa. Para o Cruzamento, é considerado apenas os pesos, que são características das soluções

escolhidas recombinadas, gerando novos indivíduos, ou seja, é realizado um cruzamento de dois pontos. A *fitness function* é o componente mais importante de qualquer algoritmo genético e levado em consideração no algoritmo. É por meio desta função que se mede quão próximo um indivíduo está da solução desejada ou quão boa é esta solução, um conjunto de teste para identificar os indivíduos mais aptos, ou mesmo uma "caixa preta"onde sabe-se apenas o formato das entradas e nos retorna um valor que se quer otimizar. Se houver pouca precisão na avaliação, uma ótima solução pode ser posta de lado durante a execução do algoritmo, além de gastar mais tempo explorando soluções pouco promissoras.

O método utiliza como critério de parada o *Brier Score* menor que 0.1, caso esse critério não fosse atendido, o número máximo de teste seria de 50 combinações, após isso, a melhor resposta seria o de menor *Brier Score* encontrado. A quantidade de 50 combinações foi definida assim, pois com as máquinas disponíveis para teste, caso a quantidade fosse maior que 50, o erro de estouro de memória acontecia com frequencia e o experimento não era concluído. Por testar no máximo 50 combinações por vez, existe a possibilidade do resultado encontrado na primeira vez não seja o ideal e o código precise ser executado mais de uma vez.

Para melhor processamento, o método genético, diferente do método força bruta, lê os dados de um arquivo contendo apenas números, a sequencia das perguntas é a mesma do método anterior. O primeiro dígito corresponde ao número de nós pais do nó filho. Os próximos números correspondem à uma sequencia de $2 * n$ conjunto de cinco dígitos, onde n é o número de nós pais e o valor da sequencia precisa ter soma 1, pois o valor 1 corresponde a 100% de confiança na resposta. Cada sequencia corresponde a uma configuração do nó, e cada número corresponde aos valores de *(Muito Baixo, Baixo, Médio, Alto, Muito Alto)* respectivamente. Seguem exemplos de configurações para a entrada dos nós no arquivo:

- 2 nós: 2 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0
- 3 nós: 3 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0
- 4 nós: 3 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0

Como o usuário tem a liberdade de definir os valores dos nós, é possível que, na dúvida, o usuário selecione mais de uma opção. No exemplo $(00.60.400)$, a soma continua sendo 1 e

o usuário delega 0.6 como valor de confiança para *Baixo* e 0.4 para *Médio*. No método força bruta o usuário só pode optar por uma configuração como resposta.

No pior caso, quando as 50 combinações precisam ser testadas, o tempo de processamento em torno de 20 a 30 minutos. A Figura 3.3.1, ilustra um exemplo de comparação de tempo de processamento e resultado ideal dos métodos força bruta e genético para nós filhos com dois nós pais. O Apêndice B, mostra a comparação de alguns testes considerando a quantidade de execução necessária para que o método genético retorne o resultado ideal. De acordo com experiementos, quatro vezes seria o número ideal, quando os primeiros resultados não são satisfatórios.

		Valores esperados					Força Bruta	Genético
N1	**N2**	**VL**	**L**	**M**	**H**	**VH**	**mean(1,2)**	**mean(9,5)**
VH	**VL**				1		8.00E-09	8.00E-09
VL	**VH**		1				6 min	30 min
VL	**M**		1					
M	**VL**		1					

		Valores esperados					Força Bruta	Genético
N1	**N2**	**VL**	**L**	**M**	**H**	**VH**	**min(2,2)**	**mean(6,3)**
VH	**VL**		1				8.00E-09	8.00E-09
VL	**VH**		1				6 min	30 min
VL	**M**		1					
M	**VL**		1					

Figura 3.9: Comparativo Força Bruta x Genético - 2 nós

3.3.2 Limitações

As limitações deste método estão relacionadas com a quantidade de execução do método. Quanto maior o número de nós pais, maior a quantidade de combinações possíveis. Apesar do método considerar as combinações escolhidas para teste como prováveis de resultado ideal, a quantidade de combinações é limitada. O método foi limitado a 50 possibilidades porque na máquina de execução, se a quantidade fosse maior, um erro de estouro de memória é mostrado e a execução é terminada.

Com a quantidade limitada de combinações, existe a possibilidade do resultado não ser

o mais indicado. Alguns dos testes realizados mostraram melhor resultado depois da primeira execução. Para um nó filho com dois nós pais, não existe esse problema, pois a maior parte das combinações são testadas, a partir disso, existe um grau de incerteza no resultado. Quando maior a quantidade de nós pais, menor seria o grau de confiança no primeiro resultado. Quando o *Brier score* é muito baixo, é garantia de um bom resultado, caso contrário, é necessário uma nova execução para conferir se existe um melhor caso.

Com relação às limitações do experimento, conclui-se que existem ameaças de conclusão à validade. É possível que, para o resultado ideal, seja necessário vários testes, porém, o método genético tem como objetivo selecionar combinações com potencial de melhor resultado. Com isso, não há nenhuma razão para acreditar que o método genético possa precisar de uma grande quantidade de repetições.

3.4 Avaliação do Método Força Bruta e Método Genético

Assim como o método regras de produção, para validar os algoritmos força bruta e genético, foi executado um experimento utilizando uma amostra de nós da rede Bayesiana, já mostrada na Figura D.4, apresentada em Perkusich *et al.* [28].

Contou-se com o conhecimento de um especialista com cinco anos de experiência usando nós ranqueados, para ao questionário similar ao do primeiro método. As respostas foram analisadas utilizando ambos os métodos e comparada conforme a Figura 3.3.1, mostra, para dois nós pais.

Percebe-se que para ambos os métodos, o *Brier score* é muito baixo, garantindo a confiança na resposta. Para um nó filho com dois nós pais, considera-se bem mais vantajoso utilizar o método força bruta, pois o mesmo leva apenas 6 minutos para garantir o melhor resultado. O método genético, apensar de garantir o melhor resultado na primeira execução, uma vez que não existem muitas possibilidades de combinações, demora meia hora.

Para um nó filho com três pais, os resultados são diferentes, porém o valor do *Brier score* continua satisfatório. A Figura 3.10 mostra que o método força bruta demora uma hora e quarenta minutos para a execução, enquanto o método genético demora 20 minutos. Em dois casos, o resultado do *Brier score* foi ideal no primeiro caso, porém no pior caso analisado, esse melhor caso só foi possível na quarta execução. Mesmo que seja necessário

aplicar o método genético quatro vezes, o tempo de espera ainda é menor que o método força bruta. Se o especialista considera tempo um fator importante na usabilidade do método, o genético seria o mais aconselhável nesse caso.

			Valores esperados					Força Bruta	Genético	Método Genético - Tentativas
N1	N2	N3	VL	L	M	H	VH	wmin(1,4,2)	wmin(1,5,3)	Tentativa 1 = wmin(1,10,2)
VH	VL	VL		1				8.66679e^-009	8.66679e^-010	0.666633
VL	VH	VL		1				1h40min	20min	Tentativa 2 = wmean(7,9,5)
VL	VL	VH		1						0.999933
VL	VH	VH				1				Tentativa 3 = wmean(7,5,8)
VH	VL	VH		1						0.666667
VH	VH	VL			1					

			Valores esperados					Força Bruta	Genético	Método Genético - Tentativas
N1	N2	N3	VL	L	M	H	VH	mean(1,6,2)	mean(1,7,3)	X
VH	VL	VL	1					8.66679e^-009	8.66679e^-010	
VL	VH	VL				1		1h40min	20 min	
VL	VL	VH		1						
VL	VH	VH					1			
VH	VL	VH		1						
VH	VH	VL				1				

			Valores esperados					Força Bruta	Genético	Método Genético - Tentativas
N1	N2	N3	VL	L	M	H	VH	mean(1,10,5)	wmean(1,10,3)	Tentativa 1 = wmean(1,9,4)
VH	VL	VL	1					0.165137	0.333333	0.333333
VL	VH	VL			1			1h40min	20min	
VL	VL	VH		1						
VL	VH	VH					1			
VH	VL	VH		1						
VH	VH	VL				1				

			Valores esperados					Força Bruta	Genético	Método Genético - Tentativas
N1	N2	N3	VL	L	M	H	VH	mean(7,7,2)	mean(9,7,2) T1	X
VH	VL	VL			1			0.332537	0.666633	
VL	VH	VL			1			1h40min	20 min	
VL	VL	VH			1					
VL	VH	VH			1					
VH	VL	VH			1					
VH	VH	VL				1				

Figura 3.10: Comparativo Força Bruta x Genético - 3 nós

Para um nó filho de quatro pais, o método genético também opera entre 20 e 30 minutos, porém, para quatro pais, a quantidade de combinações de funções e pesos é bem maior e esse fator é de risco para ambos os métodos. Com relação ao método genético, as 50 combinações como condição de parada pode não ser suficiente e seja necessário até mais de quatro execuções para um resultado ideal. Com relação ao método força bruta, a situação é

mais crítica, pois a máquina utilizada para os experimentos não possui memória suficiente para a quantidade de processamento exigida. Dependendo da memória da máquina, até o método para três nós pode reportar memória insuficiente.

Sendo assim, os métodos são garantia de um resultado confiante, pois retornaram um *Brier score* ótimo para todos os casos analisados, cabe e ao usuário decidir qual o melhor método para utilizar. É preciso analisar os fatores tempo e disponibilidade de memória da máquina de execução dos métodos.

Capítulo 4

Trabalhos Relacionados

Existem métodos para construção de TPC utilizando abordagens diferentes, como o método de Podofilline *et al.* [36], que popula TPC incompletas utilizando o método de interpolação funcional, no qual um especialista analisa dados existentes na tabela e aproximam funções para completude da mesma. O método de Wisse *et al.* [48] já é baseado em interpolação linear por partes, no qual cada nó influencia um fator de configuração particular do nó pai.

O método de Cain [20] funciona como uma calculadora, deriva dois fatores interpolares associados à mudar a probabilidade dos nós com avaliação mais alta e mais baixa. Em particular, quando a avaliação de um nó aumenta (Baixo para Médio), a probabilidade do nó de avaliação mais baixo diminui, ou se mantém, e a probabilidade do nó de avaliação mais alta (Alto) aumenta, ou se mantém.

Existem diversas abordagens distintas para o cálculo das TPC, porém, a maioria delas precisa de um especialista que analise os dados e indique as funções de probabilidade condicionais. Os métodos propostos têm como objetivo diminuir essa dependência.

4.1 Using Ranked Nodes to Model Qualitative Judgments in Bayesian Networks

Em Fenton *et al.* [31], foi apresentada a utilização de uma escala ordinal para representar variáveis aleatórias em Redes Bayesianas. A utilização de nós ordinais em redes Bayesianas justifica-se devido a necessidade de especialistas utilizarem esquemas simples de médias

para definir a tendência central de nós filhos baseando-se no conjunto de valores causais dos nós pais.

Além disso, foi apresentada uma solução baseada em distribuição Normal truncada [13] para construir uma TPC com duas variáveis: σ^2 e μ. σ^2 é a variância de erro que representa a incerteza dos resultados. μ é o índice de credibilidade, representado com valores reais. μ representa a contribuição de cada nó filho, X_i, no nó pai, *Y*. Há quatro tipos de equações ponderadas para representar μ: *WMEAN, WMIN, WMAX* e *MIXMINMAX*. Apresenta-se a definição das equações ponderadas, respectivamente, nas Equações 1, 2, 3 e 4.

1. $WMEAN = mean_i =_{1,\ldots,n}= \left[\frac{\sum_{i=1}^{n} w_i X_i}{sum_{i=1}^{n} w_i}\right]$.
2. $WMIN = min_i =_{1,\ldots,n}= \left[\frac{w_i X_i + \sum_{i \neq j}^{n} X_j}{w_i + (n-1)}\right]$.
3. $WMAX = max_i =_{1,\ldots,n}= \left[\frac{w_i X_i + \sum_{i \neq j}^{n} X_j}{w_i + (n-1)}\right]$.
4. $WMINMAX =_{1,\ldots,n}= \left[\frac{WMIN(X) + WMAX(X)}{WMIN + WMAX}\right]$.

Por outro lado, a solução apresentada em Fenton *et al.* [31] não apresenta detalhes necessários para, na prática, construir TPC de nós ordinais a partir das equações ponderadas apresentadas. Por exemplo, não é apresentado como transformar a distribuição Normal truncada gerada em uma TPC tradicional. Além disso, não há informações acerca do algoritmo utilizado para gerar e misturar distribuições Normal truncadas, tamanho das amostras e iterações de simulações necessárias para ter resultados aceitáveis. Nesta pesquisa, busca-se explorar estes problemas com o intuito de descobrir como, na prática, utilizar as equações apresentadas em Fenton *et al.* [31] para construir TPC de nós ranqueados.

4.2 A procedure to detect problems of processes in software development projects using Bayesian networks

Em Perkusich *et al.* [35], foi apresentado um modelo probabilístico para ajudar *Scrum Masters* aplicar *Scrum* nas organizações. O objetivo do modelo é fornecer informações do projeto para o *Scrum Master* para ajudá-lo a estar ciente dos problemas do projeto e ter informações suficientes para orientar a equipe e melhorar as chances do projeto de sucesso. De acordo

com Torkar *et al.* [42], vários pesquisadores têm usado redes Bayesianas para modelar incertezas sobre projetos de software e têm sido bem sucedido. Por isso, foi decidido criar um modelo de projeto de desenvolvimento de software baseado em *Scrum* usando uma rede Bayesiana, que, aparentemente, não foi feito antes.

Para definir as TPC, foi feita análise estatística dos dados coletados e utilizado da informação para gerar expressões que preenchem automaticamente TPC. Finalmente, o modelo foi validado incrementalmente. Cada TPC foi testada e foi comparado os resultados reais com os resultados esperados. A validação do modelo foi feita, aplicando-se cenários e verificando se o modelo seria útil para um *Scrum Master* identificar áreas no projeto que precisam ser melhoradas.

Uma TPC pode ser representada por *Pr* = *A*|*B*, onde *A* é uma variável dependente e *B* é um conjunto de nós pais. Assim, o conjunto *P* = $\{P_{ri}, \ldots, P_{r|}P|\}$ representa todas as TPC para a rede Bayesiana. Dessa forma, o problema foi encontrar todos os elementos do conjunto de *P*. Para encontrá-los, para cada P_{ri} onde $1 \leq i \leq |P|$, foi preciso quantificar a relação entre *A* e B_j, onde $1 \leq j \leq e|B| \in B_j$.

Para que os resultados da pesquisa criassem diretamente as expressões matemáticas, foi utilizado a escala de Likert de 5 pontos, variando de (*Very Low Inuence*) para (*Very High Inuence*). Consequentemente, no lugar dos nós de entrada para criar diretamente as expressões matemáticas para preencher as TPC, os resultados da pesquisa retornam, para cada questão, uma lista ordenada das relações por sua magnitude relativa. As listas ordenadas foram usadas para criar expressões matemáticas para preencher as TPC.

Para definir as TPC, assumiu-se que os nós são ranquados e usa-se AgenaRisk, característica de ter uma distribuição normal truncada (distribuição TNormal) para nós ranquados. A distribuição TNormal é caracterizada por dois parâmetros: média e variância. A média é calculada por uma expressão ponderada que recebe, do nó dado, infuência dos nós pais. A variação reflete a confiança no resultado. Assume-se que a confiança é alta e constante. Dado isso, parte-se da variância para todos os nós como $5.0E^{-4}$, que é o valor mínimo possível para definir a variância na AgenaRisk. AgenaRisk suporta quatro tipos de expressões ponderada: média ponderada, mínimo ponderado, máximo ponderado e misturar mínimo-máximo. Para todos os nós, utilizou-se a média ponderada (*WMEAN*) porque assume-se que não é o caso de que um nó pai é essencial inclinar a tendência central no nó ranqueado para

qualquer um dos possíveis valores extremos no modelo. Para definir os pesos nas expressões *WMEAN*, foram utilizados os resultados da análise estatística dos resultados da pesquisa. Para cada pergunta na pesquisa, foi gerado uma expressão ponderada para preencher automaticamente uma TPC. Foi criado um algoritmo para traduzir a lista ordenada de magnitude relativa de relações em uma expressão ponderada, mostrado no algoritmo abaixo.

```
Data: n = the number of relationships
Result: Weighted function to create a NPT
for i ← n to 0 do
    Get the relationship with greatest relative
    magnitude;
    Add the relationship to the function with the
    weighted value of i;
    if there are any more relationships with the same
    relative magnitude then
        m = the number of additional relationships with
        the same relative magnitude;
        current section becomes this one;
        for j ← m to 0 do
            Add the relationship to the function with the
            weighted value of i
        end
        Decrement i by the number of additional
        relationships you added to the function;
    end
end
```

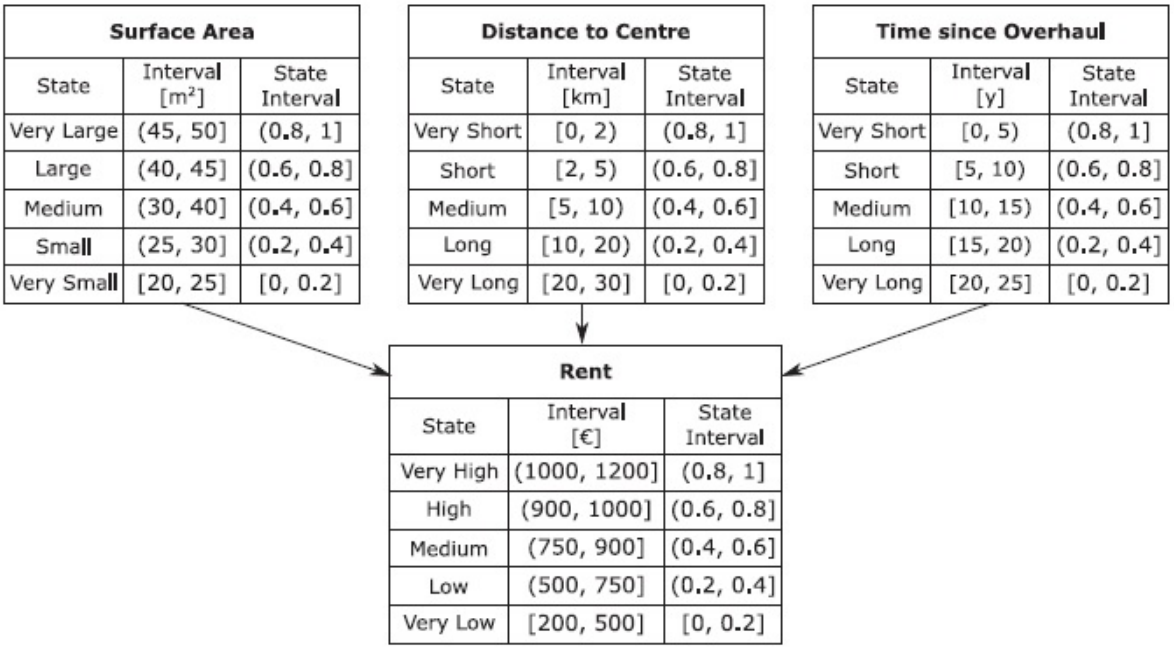
Figura 4.1: Algoritmo 1

4.3 Improving Construction of Conditional Probability Tables for Ranked Nodes in Bayesian Networks

Laitila *et al.* [27] elaborou o método nós ranqueados (MNR), que é utilizado para a construção de TPC, para redes Bayesianas constituídas por uma classe de nós chamados nós ranqueados. Tais nós normalmente representam a falta de quantidade contínua bem estabelecida das escalas intervalares e, portanto, são expressas por escalas ordinais. Com base no levantamento do especialista, a TPC do nó filho é gerada no MNR agregando estados ponderados de nós pais com uma expressão de peso. MNR é também aplicado aos nós que são expressos por meio de escalas intervalares. No entanto, a utilização do método desta forma pode ser ineficaz devido aos desafios que não são abordados na literatura existente. Para

superar os desafios, o método apresenta uma nova abordagem que facilita o uso de MNR. Ela consiste em orientações relativas à discretização das escalas intervalares para escalas ordinais e a determinação de uma expressão de peso com base em avaliações do especialista sobre o nó filho.

Primeiro, há uma diretriz para discretizar as escalas intervalares à compatibilidade com o funcionamento do MNR, um exemplo dessa discretização é ilustrada na Figura 4.2. Em segundo lugar, há uma diretriz para determinar a expressão dos peso dos nós pais por meio de avaliações do especialista sobre o nó filho em vários cenários, definidos nas escalas intervalares dos nós. A determinação é baseada em interpretações e condições de viabilidade dos pesos que são derivados do papel para cada expressão de peso. Em terceiro lugar, há sugestões de maneiras relativas ao refinamento da TPC após a sua verificação.



Surface Area

State	Interval [m²]	State Interval
Very Large	(45, 50]	(0.8, 1]
Large	(40, 45]	(0.6, 0.8]
Medium	(30, 40]	(0.4, 0.6]
Small	(25, 30]	(0.2, 0.4]
Very Small	[20, 25]	[0, 0.2]

Distance to Centre

State	Interval [km]	State Interval
Very Short	[0, 2)	(0.8, 1]
Short	[2, 5)	(0.6, 0.8]
Medium	[5, 10)	(0.4, 0.6]
Long	[10, 20)	(0.2, 0.4]
Very Long	[20, 30]	[0, 0.2]

Time since Overhaul

State	Interval [y]	State Interval
Very Short	[0, 5)	(0.8, 1]
Short	[5, 10)	(0.6, 0.8]
Medium	[10, 15)	(0.4, 0.6]
Long	[15, 20)	(0.2, 0.4]
Very Long	[20, 25]	[0, 0.2]

Rent

State	Interval [€]	State Interval
Very High	(1000, 1200]	(0.8, 1]
High	(900, 1000]	(0.6, 0.8]
Medium	(750, 900]	(0.4, 0.6]
Low	(500, 750]	(0.2, 0.4]
Very Low	[200, 500]	[0, 0.2]

Figura 4.2: Discretização de uma rede Bayesiana

Essa abordagem responde aos desafios reconhecidos no MNR. Seguindo a diretriz de discretização, uma base é definida para a geração de TPC sensatas com MNR. Por sua vez, a orientação para a determinação de uma expressão de peso fornece uma maneira transparente e compreensível para elicitá-los do especialista. As interpretações dos pesos facilitam o levantamento no MNR de uma forma semelhante às interpretações dos critérios dos pesos para facilitar o levantamento no contexto de multi-critério de análise de decisão. Além disso,

para um especialista envolvido no trabalho diário com um fenômeno ou um sistema descrito, como uma rede Bayesiana, a consideração do do nó filho em vários cenários pode ser uma forma familiar de raciocínio. Isso pode diminuir a tensão cognitiva colocada sobre o especialista na eliciação. Como o modo de avaliação é feito com dados intervalares, o grupo de elicitação é também apto à formação dos intervalos com base nos pontos de vista dos vários especialistas.

Abaixo a Figura 4.3 ilustra uma breve comparação entre os métodos apresentados e os principais trabalhos relacionados.

	Fenton	Perkusich	Pekka	Regras de Produção	Força Bruta	Genético
Diminuição da dependência de um especialista para definição das funções de probabilidade condicional		X		X	X	X
Considera todas as funções de probabilidade condicional	X		X	X	X	X
Considera uma escala intervalar de acordo com especialistas			X			

Figura 4.3: Comparação dos métodos propostos com os principais trabalhos relacionados

Capítulo 5

Considerações Finais

Nesta dissertação de mestrado, abordou-se a problemática relacionada à automatização da definição de nós ranqueados, dado que o conhecimento de um especialista foi elicitado, além de encapsular o conhecimento de baixo nível necessário para utilizar nós ranqueados. O trabalho é relevante, pois definir TPC é um trabalho de esforço exponencial e esse tipo de estudo tem sido o foco de vários trabalhos passados, que apesar dos avanços, mantiveram alguns problemas inerentes à essa atividade.

Neste trabalho, propõem-se três métodos para abordar o problema: um sistema especialista para, com o conhecimento suscinto do especialista de domínio, automatizar a definição de TPC de redes Bayesianas; um método que utiliza força bruta; e um terceiro baseado em algoritmos genéticos. Os métodos são baseados em nós ranqueados e diminuem a complexidade da definição das TPC. Além disso, aumentam a aplicabilidade da utilização de Rede Bayesiana, porque escondem do especialista de domínio a complexidade sobre a calibração da TPC.

A avalidação realizada demonstrou que as abordagens melhoraram o método de definição das funções de probabilidade condicionais do modelo apresentado em Perkusich *et al.* [28], automatizando a abordagem apresentada em Fenton *et al.* [31].

5.1 Limitações

O método força bruta requer um maior espaço de memória para execução de casos com nós filhos com número de nós pais maior que três. Como o método força bruta testa muitas

combinações, além do tempo de processamento, a utilização de memória aumentam consideravelmente. O método genético não possui esse problema de memória pois tem um número fixo de combinações para teste, porém, como todos os casos não são testados, existe a possibilidade de precisar utilizar o algoritmo várias vezes.

Os testes utilizam a ferramenta AgenaRisk [1], que não é gratuita, portanto, para testes futuros, é preciso renovar o contrato da licença, que tem duração de um ano.

5.2 Trabalhos Futuros

Para trabalhos futuros, pretende-se investigar os riscos sobre o uso de escalas ordinais para obter conhecimentos de especialistas e usar a lógica Fuzzy para modelar os dados elicitados a partir de especialistas. É preciso considerar a incerteza do especialista na hora de elicitação dos dados, esta pode ser outra variável influente para um método futuro.

Bibliografia

[1] Agenarisk 6.1. `http://www.agena.co.uk`, note = Agena Ltd.

[2] Expertsinta. http://www.lia.ufc.br/ bezerra/exsinta/. Agena Ltd.

[3] M. A. Arbib. *The Handbook of Brain Theory and Neural Networks*. MIT Press, MA, USA, 1 edition, 1995.

[4] I. Ben-Gal. Bayesian networks. In F. Falti F. Ruggeri and R. Kenett, editors, *Encyclopedia of Statistics in Quality & Reliability*. Wiley & Sons, 2007.

[5] MIRKO BEZERRA, J. ; Renan Willamy ; PERKUSICH. Um método para a construção de funções de probabilidade de redes bayesianas baseada em nós ranqueados. In *X CONNEPI*, 2015.

[6] E. Horvitz D. Heckerman and B. Nathwani. Toward normative expert systems: Part i. the pathfinder project. In *Methods Inform Med.*, volume 31, pages 90 – 105, 1992.

[7] J. Breese D. Heckerman and K. Rommelse. Decision-theoretic troubleshooting. In *Commin. ACM*, volume 38, pages 49 – 57, 1995.

[8] A. Groso D. Pluss and T. Meyer. Expert judgements in risk analysis: A strategy to overcome uncertainties. In *Chem. Eng. Trans.*, volume 31, pages 307 – 312, 2013.

[9] Possamai O. Dalla Valentina L. V. de Oliveira, M. A. Applying bayesian networks to performance forecast of innovation projects: A case study of transformational leadership influence in organizations oriented by projects. In *Expert Systems with Applications*, volume 39, pages 5061 – 5070, 2012.

[10] F. Diez. Oparameter adjustment in bayes networks. the generalized noisy or-gate. In *Proc. 9th Conf. Uncertainty Artif. Intell*, pages 99 – 105, 1993.

[11] M. Druzdel and L. van der Gaag. Building probabilistic networks:" where do the numbers come from? In *IEEE Trans. Knowl. Data Eng.*, pages 481 – 483, 2000.

[12] G. Pai E. Denney and I. Habli. Towards measurement of confidence in safety cases. In *Proc. 5th Int. Symp. Empirical Softw. Eng. Meas*, pages 380 – 383, 2011.

[13] F. Cozman e E. Krotkov. Truncated Gaussians as Tolerance Sets. *Technical Report CMU-RI-TRI*. PhD thesis, Carnegie Mellon University, 1997.

[14] Y. Park F. Lee and J. G. Shin. Large engineering project risk management using a bayesian belief network. In *Expert Syst. Appl.*, volume 36, pages 5880 – 5887, 2009.

[15] L. Falzon. Using bayesian network analysis to support centre of gravity analysis in military planning. In *Proc. Winter Simul. Conf.*, volume 170, pages 629 – 643, 2006.

[16] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian networks classifiers. In *Machine Learning*, volume 29, pages 131 – 163, 1997.

[17] S.-C. Jang H. Son H.-S. Eom, G.-Y. Park and H. Kang. Vvbased remaining fault estimation model for safety–critical software of a nuclear power plant. In *Ann. Nucl. Energy*, volume 51, pages 38 – 49, 2013.

[18] D. Heckerman. *A tutorial on learning with Bayesian networks*. Microsoft Advanced Technology Division, Microsoft Corporation, 1995.

[19] R. A. Howard and J. E. Matheson. Inflience diagrams. In *Decision Anal.*, volume 2, pages 127 – 143, 2005.

[20] Cain J. Planning improvements in natural resource management. In *using Bayesian networks to support the planning and management of development programmes in the water sector and beyond*, 2001.

[21] J. Poporudas J. Pousi and K. Virtanen. Simulation metamodeling with bayesian networks. In *J. Simul.*, volume 7, pages 297 – 311, 2011.

[22] J.Pearl and S. Russel. Bayesian networks. In M. Arbib, editor, *Handbook of Brain Theory and Neural Networks*, pages 157 – 160. MIT Press, 2001.

[23] F. Jensen B. Falck S. Andreassen K. Olesen, U. Kjaerulff and S. Anderson. A munin network for the median nerve - a case study on loops. In *Appl. Arfif. Intell. Int. J.*, volume 3, pages 385 – 403, 1989.

[24] T. Raivio K. Virtanen and R. P. Hamalainen. Modeling pilot's sequential maneuvering decisions by a multistage influence diagram. In *J. Guid., Control, Dyn.*, volume 27, pages 665 – 677, 2004.

[25] C. Witteman B. Aleman L. Van der Gaag, S. Renooij and B. Taal. Probabilities for a probabilistic network: A case-study in oesophageal carcinoma. In *Artif. Intell. Med.*, pages 123 – 148, 2002.

[26] Pekka Laitila. *Improving the Use of Ranked Nodes in Elicitation of Conditional Probabilities for Bayesian Networks*. 2013.

[27] Pekka Laitila. Improving construction of conditional probability tables for ranked nodes in bayesian network. In *IEEE Trans. Knowl. Dava Eng.*, volume 28, pages 1691 – 1705, 2016.

[28] H. O. de Almeida M. Perkusich and A. Perkusich. A model to detect problems on scrum-based software development projects. In *In Proceedings of the 28th An- nual ACM Symposium on Applied Computing, SAC'13*, pages 1037 – 1042, New York, NY, USA, 2013.

[29] D. Mayara. Regras de produção na representação do conhecimento.

[30] S. Monti and G. Carenini. Dealing with the expert inconsistency in probability elicitation. In *IEEE Trans. Knowl. Data Eng.*, pages 499 – 508, 2000.

[31] M. Neil N.E. Fenton and J.S. Caballero. Using ranked nodes to model qualitative judgments in bayesian networks. In *Proc. of IEEE Transactions on Knowledge and Data Engineering*, pages 1420 – 1432, 2007.

[32] D. Nikovski. Constructing bayesian network for medival diagnosis from incomplete and partially correct statistics. In *IEEE Trans. Knowl. Dava Eng.*, volume 12, pages 509 – 516, 2000.

[33] J. Pearl. Probabilistic reasoning in intelligent systems: Networks of plausible inference. 1988.

[34] Mirko Perkusich, Angelo Perkusich, and Hyggo Almeida. Using survey and weighted functions to generate node probability tables for *Bayesian* networks. In *Proceedings of BRICS-CCI 2013*, 2013.

[35] Mirko Perkusich, Gustavo Soares, Hyggo Almeida, and Angelo Perkusich. A procedure to detect problems of processes in software development projects using bayesian networks. In *Expert Systems with Applications*, volume 42, pages 437 – 450, 2015.

[36] Dang VN. Podofillini L, Mkrtchyan L. Aggregating expert-elicited error probabilities to build hra models. In *Proceedings of ESREL*, pages 14 – 18, 2014.

[37] J. Poporudas and K. Virtanen. Analyzing air combat simulation results with dynamic bayesian networks. In *Eur. J. Oper. Res.*, pages 1370 – 1377, 2007.

[38] J. Poporudas and K. Virtanen. Simulation metamodeling with dinamic bayesian networks. In *Eur. J. Oper. Res.*, volume 214, pages 644 – 655, 2011.

[39] C. Preston and A. Colman. Optimal number of response categories in rating scales: Reliability, validity, discriminating power, and respondent preferences. In *Acta Psychologica*, volume 104, pages 1 – 15, 2000.

[40] A. Qazi, J. Quigley, A. Dickson, B. Gaudenzi, and Ş Ö Ekici. Cost and benefit analysis of supplier risk mitigation in an aerospace supply chain. In *Industrial Engineering and Systems Management (IESM), 2015 International Conference on*, pages 850–857, Oct 2015.

[41] Renata Saraiva Arthur Freire Hyggo Almeida Raissa da Silva, Mirko Perkusich and Angelo Perkusich. Improving the applicability of bayesian networks through production rules, conference on software engineering and knowledge engineering. volume 214, pages 644 – 655, 2016.

[42] Adnan Alvi Wasif Afzal Richard Torkar, Nasir Awan. Predicting software test effort in iterative development using a dynamic bayesian network. In *International Symposium on Software Reliability Engineering*.

[43] R.Neapolitan. *Learning Bayesian Networks*. Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2004.

[44] S. Russell and P. Norvig. *Artificial Intelligence: A modern Approach*. Englewood Cliffs, NJ, USA: Prentice Hall, 2003.

[45] R. L. Sousa and H. H. Einstein. Risk analysis during tunnel construction using bayesian networks: Porto metro case study. In *Tunnelling Underground Space Technol.*, volume 27, pages 86 – 100, 2012.

[46] L. Uusitalo. Advantages and challenges of bayesian networks in environmental modelling. In *Ecological Modelling*, volume 203, pages 312 – 318, 2007.

[47] WINSTON. page chapter 7, 1992.

[48] van Elst NP Barros AI Wisse BW, vanGosliga SP. Relieving the elicitation burden of bayesian belief networks. In *Proceedings of the sixth Bayesian modelling applications workshop on UAI*, 2008.

[49] Wu X. Ding L. Skibniewski M. J. Yan Y. Zhang, L. Decision support analysis for safety control in complex project environments based on bayesian networks. In *Expert Systems with Applications*, volume 40, pages 4273 – 4281, 2013.

Apêndice A

Tabelas de Combinações da Coleta de Dados do Especialista para o Método Regras de Produção

VL-VH	VH-VL	VL-M	M-VL	Probable Answers
L	L	VL	VL	MIN(3,3)* MIN(3,4) **MIN(3,5)** MIN(4,3)* MIN(4,4) MIN(4,5) MIN(5,3) MIN(5,4)
L	L	VL	L	MIN(3,2) MIN(4,2) MIN(3,3)* MIN(4,3)* **MIN(5,1)** MIN(5,2) MIN(5,3)
L	L	VL	M	X
L	L	L	VL	MIN(2,3) **MIN(2,5)** MIN(3,3)* MIN(3,4)* MIN(3,5)*
L	L	L	L	MIN(2,2) **MIN(2,3)** MIN(3,2)* MIN(3,3)*
L	L	L	M	X
L	L	M	VL-L-M	X
L	M	VL	VL	X
L	M	VL	L	MIN(3,1) MIN(4,1)
L	M	VL	M	X
L	M	L	VL	MIN(2,4)
L	M	L	L	MEAN(5,3) MIN(2,1) **MIN(3,1)**

L	M	L	M	X
L	M	M	VL	X
L	M	M	L	X
L	M	M	M	MAX(5,5)
L	H	VL	VL	X
L	H	VL	L	MEAN(3,1)*
L	H	VL	M	MEAN(3,1)* MEAN(4,1) **MEAN(5,1)**
L	H	L	VL	X
L	H	L	L	**MEAN(2,1)** MEAN(3,1)* MEAN(4,2) MEAN(5,2) MEAN(5,3)* MAX(2,1) MAX(2,2) MAX(2,3) MAX(3,1)* MAX(3,2)* MAX(3,3)*
L	H	L	M	MEAN(3,1)* MAX(3,1)* MAX(3,2)* MAX(3,3)* MAX(4,1) MAX(4,2) MAX(4,3) MAX(5,1) **MAX(5,2)** MAX(5,3)
L	H	M	VL	X
L	H	M	L	**MAX(2,3)** MAX(2,4) MAX(2,5) MAX(3,3)* MAX(3,4)*

				MAX(3,5)*
L	H	M	M	MAX(3,3)* MAX(3,4)* MAX(3,5)* MAX(4,3) MAX(4,4) MAX(4,5) **MAX(5,3)** MAX(5,4)
M	L	VL	VL-L-M	X
M	L	L	VL	MIN(1,3)* MIN(1,4) **MIN(1,5)**
M	L	L	L	**MEAN(3,5)*** MIN(1,2) MIN(1,3)*
M	L	L	M	X
M	L	M	VL-L-M	X
M	M	VL	VL	X
M	M	VL	L	MIN(3,1) **MIN(4,1)**
M	M	VL	M	X
M	M	L	VL	X
M	M	L	L	**MEAN(1,1)** MEAN(2,2) MEAN(2,3) MEAN(3,2) MEAN(3,3) MEAN(3,4) MEAN(3,5)* MEAN(4,3) MEAN(4,4) MEAN(4,5) MEAN(5,3)* MEAN(5,4) MEAN(5,5) MAX(1,2)

				MAX(1,3) MIN(2,1) MIN(3,1)
M	M	L	M	X
M	M	M	VL	X
M	M	M	L	MAX(1,4) MAX(1,5)
M	M	M	M	MAX(5,5)
M	H	VL	VL-L-M	X
M	H	L	VL	x
M	H	L	L	**MEAN(5,3)*** MAX(2,1) MAX(3,1)*
M	H	L	M	MAX(3,1) MAX(4,1) **MAX(5,1)**
M	H	M	VL-L-M	X
H	L	VL	VL-L-M	X
H	L	L	VL	MEAN(1,3)*
H	L	L	L	MEAN(1,2)69 MEAN(1,3)* MEAN(2,4) MEAN(2,5) **MEAN(3,5)***
H	L	L	M	X
H	L	M	VL	MEAN(1,3)* MEAN(1,4) **MEAN(1,5)**
H	L	M	L	MEAN(1,3)*
H	L	M	M	X
H	M	VL	VL-L-M	X
H	M	L	VL	X

H	M	L	L	MEAN(3,5)* **MAX(1,2)** MAX(1,3)
H	M	L	M	X
H	M	M	VL	X
H	M	M	L	**MAX(1,4)** MAX(1,5)
H	M	M	M	X
H	H	VL	VL-L-M	X
H	H	L	VL	X
H	H	L	L	MAX(2,2) MAX(2,3)69 MAX(3,2)* MAX(3,3)*
H	H	L	M	MAX(3,2)* MAX(3,3)* MAX(4,2)59 MAX(4,3) **MAX(5,2)** MAX(5,3)
H	H	M	VL	X
H	H	M	L	MAX(2,3)69 MAX(2,4) MAX(2,5) MAX(3,3)* MAX(3,4) MAX(3,5)
H	H	M	M	MAX(3,3)* MAX(4,4) MAX(4,5) **MAX(5,3)** MAX(5,4)

VL-VH-VL	VH-VH-VL	VH-VL-VH	VL-VH-VH	VL-VL-VH	VH-VL-VL	RESULT
H	VH	H	H	L	H	WMAX(5,5,1)
H	VH	H	H	M	H	WMAX(5,5,3)
H	VH	VH	VH	H	H	WMAX(5,5,5)
H	H	L	H	L	L	WMEAN(1,5,1)
H	H	M	X	X	X	X
H	H	H	H	L	L	WMAX(1,5,1)
H	H	H	H	L	M	WMAX(3,5,1)
H	H	H	H	L	H	WMAX(5,1,1)
H	H	H	H	M	L	WMAX(1,5,3)
H	H	H	H	M	M	WMAX(3,5,3)
H	H	H	VH	H	L	WMAX(1,5,5)
H	M	L-M-H	X	X	X	X
M	VH	M	M	VL	M	WMEAN(5,5,1)
M	VH	M	H	VL	L	WMEAN(3,5,1)
M	H	M	M	L	M	WMEAN(5,5,3)
M	H	M	H	L	L	WMEAN(1,2,1) **WMEAN(3,5,3)** WMEAN(1,3,1)
M	H	M	H	L	M	WMEAN(3,3,1)
M	H	M	VH	L	VL	WMEAN(1,5,3)
M	H	H	H	L	L	**WMAX(1,2,1)** WMAX(1,3,1)
M	H	H	H	L	M	WMAX(3,3,1)
M	H	H	H	L	H	WMAX(5,3,1) WMAX(5,3,3)
M	H	H	H	M	L	WMAX(1,1,3)

						WMAX(1,3,3)
M	H	H	H	M	M	WMAX(3,3,3)
M	H	H	H	H	L	WMAX(1,3,5)
M	M	M	H	L	L	WMEAN(1,2,2)
M	M	M	H	M	L	WMEAN(1,3,3) WMIN(1,2,2)
M	M	M	VH	M	VL	WMEAN(1,5,5)
M	M	H	H	M	L	WMEAN(1,1,3)
L	VH	H	M	VL	M	WMEAN(5,3,1)
L	H	L	L	VL	L	WMIN(5,5,1)
L	H	L	M	L	L	WMIN(3,5,1) WMIN(1,5,1) **WMIN(1,2,1)**
L	H	M	L	L	L	WMIN(5,1,1)
L	H	M	M	L	L	WMIN(3,3,1)
L	H	M	H	L	L	WMIN(1,3,1)
L	H	H	L	L	H	WMEAN(5,1,1)
L	H	H	M	L	L	WMIN(3,1,1) **WMIN(2,1,1)**
L	H	H	M	L	M	WMEAN(3,1,1) **WMEAN(5,3,3)**
L	H	H	H	L	L	WMEAN(5,5,5) WMEAN(3,3,3) WMEAN(1,1,1)
L	H	H	H	L	M	WMAX(3,1,1) **WMAX(2,1,1)** **WMEAN(2,1,1)**
L	H	H	H	L	H	WMAX(5,1,1)
L	H	H	H	M	L	WMAX(1,2,2) WMAX(1,1,2)

L	H	H	H	M	M	WMAX(3,1,3)
L	H	H	H	M	H	WMAX(5,1,3)
L	H	H	H	H	L	WMAX(1,1,5)
L	H	H	H	H	M	WMAX(3,1,5)
L	H	VH	H	H	H	WMAX(5,1,5)
L	M	L	L	VL	L	WMIN(5,5,3)
L	M	L	L	L	L	WMIN(5,3,3)
L	M	L	M	L	L	WMIN(3,5,3)
L	M	M	VH	M	VL	WMEAN(1,3,5)
L	M	M	H	L	L	WMIN(1,5,3) **WMIN(1,2,2)**
L	M	M	M	L	L	WMIN(3,3,3)
L	M	H	L	L	L	WMIN(5,1,3)
L	M	H	M	L	L	WMIN(3,1,3)
L	M	H	M	M	M	WMEAN(3,1,3)
L	M	H	H	L	L	WMIN(1,1,3) **WMIN(1,1,2)**
L	M	H	H	M	L	WMEAN(1,1,2)
L	L	L	H	L	VL	WMIN(1,5,5)
L	L	M	H	L	L	WMIN(1,3,5)
L	L	L	M	L	L	WMIN(3,1,5)
L	L	H	H	L	L	WMIN(1,1,5)
L	L	H	H	H	L	WMEAN(1,1,5)
VL	H	**VH**-H	M	L	M	WMEAN(5,1,3)
VL	M	**VH**-H	M	L	M	WMEAN(5,1,5)
VL	M	**VH**-H	H	M	L	WMEAN(3,1,5)
VL	L	L	L	VL	VL	WMIN(5,5,5)

VL	L	H	L	L	L	WMIN(5,1,5)

VH-VH-VL-M	VL-VL-VH-VH	M-VH-H-M	RESULT
L	L	L	X
L	L	M	WMIN(5,5,5,5) WMIN(5,1,3,5) WMIN(3,3,5,1)
L	L	H	X
L	M	L	X
L	M	M	WMIN(1,1,5,1) WMIN(1,1,5,3)
L	M	H	WMIN(1,1,5,1)
L	H	L	X
L	H	M	WMEAN(1,1,5,3)
L	H	H	WMEAN(1,1,5,1)
M	L	L	X
M	L	M	WMIN(1,5,1,1) WMIN(5,1,1,1) WMIN(5,1,1,5) WMIN(5,5,1,5) WMIN(5,3,1,1) WMIN(1,5,1,3) WMIN(5,1,1,3)
M	L	H	WMIN(1,5,1,1)
M	M	L	X
M	M	M	WMEAN(5,5,5,5) WMEAN(5,1,3,5) WMIN(1,1,1,5) WMIN(1,1,3,5)
M	M	H	WMEAN(1,1,1,1) WMEAN(5,5,5,5) WMEAN(3,3,5,1)
M	H	L	X

VH-VH-VL-M	VL-VL-VH-VH	M-VH-H-M	RESULT
L	L	L	X
L	L	M	WMIN(5,5,5,5) WMIN(5,1,3,5) WMIN(3,3,5,1)
L	L	H	X
L	M	L	X
L	M	M	WMIN(1,1,5,1) WMIN(1,1,5,3)
L	M	H	WMIN(1,1,5,1)
L	H	L	X
L	H	M	WMEAN(1,1,5,3)
L	H	H	WMEAN(1,1,5,1)
M	L	L	X
M	L	M	WMIN(1,5,1,1) WMIN(5,1,1,1) WMIN(5,1,1,5) WMIN(5,5,1,5) WMIN(5,3,1,1) WMIN(1,5,1,3) WMIN(5,1,1,3)
M	L	H	WMIN(1,5,1,1)
M	M	L	X
M	M	M	WMEAN(5,5,5,5) WMEAN(5,1,3,5) WMIN(1,1,1,5) WMIN(1,1,3,5)
M	M	H	WMEAN(1,1,1,1) WMEAN(5,5,5,5) WMEAN(3,3,5,1)
M	H	L	X

Apêndice B

Dados Elicitados do *Scrum Master*

Node: Work validation quality

Truth Table

Sprint Review meeting achieving its goals	Stakeholders feedback outside of the sprint Review meeting	Expected
Very low	Very high	Low
Very high	Very low	High
Very low	Medium	Low
Medium	Very low	Low
Very low	High	Low
Low	High	Low
High	Low	High
Low	Medium	Low
Medium	Low	Medium
Medium	Very high	High
Very high	Medium	High
Very high	Low	High
High	Very low	Medium
Low	Very high	Medium
Medium	High	Medium
Very low	Very low	Medium

Calculated Parameters

type	wmean
W1	2
W2	1
var	0.0005

Node: Product backlog quality

Truth Table

Product backlog management	Release plan	Expected
Very low	Very high	Low
Very high	Very low	Low
Very low	Medium	Low
Medium	Very low	Low
Very high	Medium	Medium
Medium	Very high	Medium
Medium	Low	Low
Very low	High	Low
High	Very low	Low
High	Low	Low
Low	Very High	Medium
Medium	Very low	Very low
Low	Very low	Very low
Very low	Very low	Very low
Medium	Medium	Medium
High	Medium	Medium

Calculated Parameters

type	wmin
W1	2
W2	3
var	0.0005

Node: Product increment quality

Truth Table

Sprint progress	Sprint Planning quality	Expected
Very low	Very high	Low
Very high	Very low	Low
Very low	Medium	Low
Medium	Very low	Low
Very low	High	Low
Low	High	Low
High	Low	Low
Low	Medium	Low
Medium	Low	Low
Medium	Very high	Medium
Very High	Medium	Medium
Very high	Low	Low
High	Very low	Low
Low	Very high	Low
Medium	High	Medium
Very low	Very low	Very low

Calculated Parameters

type	wmin
W1	2
W2	3
var	0.0005

Node: Sprint Review meeting achieving its goals

Truth Table

Stakeholder feedback	Sprint goal check	Acceptance criteria check	Expected
Very low	Very high	Very low	Low
Very low	Very low	High	Low
Very low	Very high	Very high	Medium
Very high	Very low	Very high	Medium
Very high	Very high	Very low	Medium
Very high	Medium	Medium	Medium
Very low	Very high	Low	Low
Low	High	Very low	Low
Medium	High	Very low	Low
Very low	Very low	Medium	Very low
Very low	Very low	Very high	Low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Low
High	Low	Medium	Low
Medium	Low	Medium	Low
Medium	Very high	Very high	High
Medium	High	Very high	High

Calculated Parameters

type	wmin
W1	3
W2	3
W3	3
var	0.0005

Node: Product backlog is properly ordered

Truth Table

Considers technical dependencies	Considers business value	Considers risk	Expected
Very low	Very high	Very low	Low
Very low	Very low	Very high	Low
Very low	Very high	Very high	Medium
Very high	Very low	Very high	Low
Very high	Very high	Very low	Medium
Very high	Medium	Medium	Medium
Very low	Very high	Low	Low
Low	High	Very low	Low
Medium	High	Very low	Low
Very low	Very low	Medium	Very low
Very low	Very low	Very high	Very low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Low
Medium	Low	Medium	Low
Medium	Very high	Very high	High
Medium	High	Very high	High

Calculated Parameters

type	wmin
W1	3
W2	5
W3	3
var	0.0005

Node: Sprint backlog quality

Truth Table

Size estimation quality	Task breakdown quality	Product backlog items are properly detailed	Expected
Very low	Very high	Very low	Low
Very low	Very low	Very high	Low
Very low	Very high	Very high	Medium
Very high	Very low	Very high	Medium
Very high	Very high	Very low	Low
Very high	Medium	Medium	Medium
Very low	Very high	Low	Low
Low	High	Very low	Low
Medium	High	Very low	Low
Very low	Very low	Medium	Low
Very low	Very low	Very high	Low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Medium
Medium	Low	Medium	Medium
Medium	Very high	Very high	High
Medium	High	Very high	High

Calculated Parameters

type	wmin
W1	1
W2	2
W3	2
var	0.0005

Node: Code inspection quality

Truth Table

Static code analysis	Peer code review	Pair programming	Expected
Very low	Very high	Very low	High
Very high	Very low	Very low	High
Very low	Very low	Very high	High
Very low	Very high	Very high	High
Very high	Very low	Very high	High
Very high	Very high	Very low	High
Very high	Medium	Medium	High
Very low	Medium	Medium	High
Low	High	Very low	High
Medium	High	Very Low	High
Very low	Very low	Medium	Medium
Very low	Very low	Low	Low
Low	Low	Low	Low
Very low	Low	Medium	Medium
High	Low	Medium	High
Medium	Low	Medium	Medium
Medium	Very high	Very high	High
Medium	High	Very high	High

Calculated Parameters

type	wmax
W1	1
W2	5
W3	1
var	0.0005

Node: Release Plan

Truth Table

Launch data defined	3 to 5 top features described	Product vision quality	Expected
Very low	Very high	Very low	Low
Very high	Very low	Very low	Low
Very low	Very low	Very high	Low
Very low	Very high	Very high	Medium
Very high	Very low	Very high	Medium
Very high	Very high	Very low	Medium
Very high	Medium	Medium	Medium
Very low	Very High	Low	Low
Low	High	Very low	Low
Medium	High	Very Low	Low
Very low	Very low	Medium	Low
Very low	Very low	Very high	Low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Low
Medium	Low	Medium	Low
Medium	Very high	Very high	Medium
Medium	High	Very high	Medium

Calculated Parameters

type	wmin
W1	1
W2	2
W3	2
var	0.0005

Node: Sprint progress

Truth Table

Development team competence	Sprint items and priorities stability	Daily Scrum quality	Expected
Very low	Very high	Very low	Low
Very high	Very low	Very low	Medium
Very low	Very low	Very high	Low
Very low	Very high	Very high	Medium
Very high	Very low	Very high	High
Very high	Very high	Very low	Medium
Very high	Medium	Medium	Medium
Very low	Very High	Low	Low
Low	High	Very low	Low
Medium	High	Very Low	Low
Very low	Very low	Medium	Low
Very low	Very low	High	Low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Medium
Medium	Low	Medium	Low
Medium	Very high	Very high	Medium
Medium	High	Very high	Medium

Calculated Parameters

type	wmin
W1	1
W2	1
W3	2
var	0.0005

Node: Product Owner overall quality

Truth Table

Progress tracking	Product backlog quality	Product Owner personal characteristics	Expected
Very low	Very high	Very low	Low
Very high	Very low	Very low	Low
Very low	Very low	Very high	Low
Very low	Very high	Very high	Medium
Very high	Very low	Very high	Low
Very high	Very high	Very low	Low
Very high	Medium	Medium	Medium
Very low	Very High	Low	Low
Low	High	Very low	Low
Medium	High	Very Low	Low
Very low	Very low	Medium	Low
Very low	Very low	Very high	Low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Low
Medium	Low	Medium	Low
Medium	Very high	Very high	Medium
Medium	High	Very high	High

Calculated Parameters

type	wmean
W1	1
W2	1
W3	2
var	0.0005

Node: Process Success

Truth Table

Product increment quality	Product Owner overall quality	Work validation quality	Expected
Very low	Very high	Very low	Very low
Very high	Very low	Very low	Very low
Very low	Very low	Very high	Very low
Very low	Very high	Very high	Low
Very high	Very low	Very high	Low
Very high	Very high	Very low	Low
Very high	Medium	Medium	Medium
Very low	Very High	Low	Low
Low	High	Very low	Very low
Medium	High	Very Low	Very low
Very low	Very low	Medium	Very low
Very low	Very low	High	Very low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Low
Medium	Low	Medium	Low
Medium	Very high	Very high	Medium
Medium	High	Very high	Medium

Calculated Parameters

type	wmin
W1	1
W2	2
W3	2
var	0.0005

Node: Development team competence

Truth Table

Continuous improvement	Software engineering techniques quality	Development team teamwork skills	Expected
Very low	Very high	Very low	Low
Very high	Very low	Very low	Low
Very low	Very low	Very high	Low
Very low	Very high	Very high	High
Very high	Very low	Very high	Low
Very high	Very high	Very low	Medium
Very high	Medium	Medium	Medium
Very low	Very High	Low	Low
Low	High	Very low	Low
Medium	High	Very Low	Low
Very low	Very low	Medium	Very low
Very low	Very low	Very high	Low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Low
Medium	Low	Medium	Low
Medium	Very high	Very high	High
Medium	High	Very high	High

Calculated Parameters

type	wmean
W1	1
W2	1
W3	2
var	0.0005

Node: Software engineering techniques quality

Truth Table

Code integration frequency	Code quality	Code inspection quality	Expected
Very low	Very high	Very low	Low
Very high	Very low	Very low	Very low
Very low	Very low	Very high	Very low
Very low	Very high	Very high	Low
Very high	Very low	Very high	Low
Very high	Very high	Very low	Low
Very high	Medium	Medium	Medium
Very low	Very High	Low	Low
Low	High	Very low	Very low
Medium	High	Very Low	Low
Very low	Very low	Medium	Very low
Very low	Very low	Very high	Very low
Low	Low	Low	Low
Very low	Low	Medium	Low
High	Low	Medium	Low
Medium	Low	Medium	Low
Medium	Very high	Very high	High
Medium	High	Very high	High

Calculated Parameters

type	wmean
W1	1
W2	1
W3	2
var	0.0005

Node: Sprint planning quality

Truth Table

Team Velocity defined	Sprint length	Sprint backlog quality	Broad and realistic goal	Expected
Very high	Very high	Very low	Medium	Low
Very low	Very low	Very high	Very high	Medium
Medium	Very high	High	Medium	Medium
Very high	Very low	Very low	Very low	Low
Very low	Very high	Very low	Very low	Low
Very low	Very low	Very high	Very low	Low
Very low	Very low	Very low	Very high	Low
Very high	Very high	Very low	Very low	Low
Very low	Very low	Very high	Very high	Medium
Very low	Very high	Very low	Very high	Low
Very high	Very high	Very high	Very low	Medium
Very high	Very high	Very low	Very high	Low
Very high	Very low	Very high	Very high	Medium
Very low	Very high	Very high	Very high	Medium

Calculated Parameters

type	wmin
W1	1
W2	1
W3	5
W4	1
var	0.0005

Node: Product backlog management

Truth Table

Product backlog items are properly detailed	Product backlog is estimated	Answer the 3 basic questions	Product backlog is emergent	Expected
Very high	Very high	Very low	Medium	Medium
Very low	Very low	Very high	Very high	Low
Medium	Very high	High	Medium	High
Very high	Very low	Very low	Very low	Very low
Very low	Very high	Very low	Very low	Very low
Very low	Very low	Very high	Very low	Low
Very low	Very low	Very low	Very high	Very low
Very high	Very high	Very low	Very low	Low
Very low	Very high	Very high	Very low	Medium
Very high	Very high	Very high	Very low	High
Very high	Very high	Very low	Very high	Low
Very high	Very low	Very high	Very high	Medium
Very low	Very high	Very high	Very high	High

Calculated Parameters

type	wmin
W1	1
W2	1
W3	5
W4	1
var	0.0005

Node: Product vision quality

Truth Table

Critical attributes to satisfy customer needs described	Broad and engaging goalConsiders business value	Short and concise	Clear and stable	Expected
Very high	Very high	Very low	Medium	Low
Very low	Very low	Very high	Very high	Low
Medium	Very high	High	Medium	Medium
Very low	Very low	Very low	Very high	Very low
Very low	Very low	Very high	Very low	Very low
Very low	Very high	Very low	Very low	Very low
Very high	Very low	Very low	Very low	Very low
Very high	Very high	Very low	Very high	Medium
Very high	Very high	Very low	Very high	Medium
Very high	Very low	Very high	Very high	Medium
Very low	Very high	Very high	Very high	Medium
Very high	Very low	Very low	Very high	Low
Very high	Very low	Very high	Very low	Low
Very high	Very high	Very low	Very low	Low
Very low	Very high	Very high	Very low	Low

Calculated Parameters

type	wmin
W1	5
W2	5
W3	5
W4	5
var	0.0005

Node: Code Quality

Truth Table

Documentation	Code Refactoring	Short and concise	Test coverage analysis	Expected
Very high	Very low	Very low	Very low	Very low
Very low	Very high	Very low	Very low	Low
Very low	Very low	Very high	Very low	Low
Very low	Very low	Very low	Very high	Low
Very high	Very high	Very low	Medium	Medium
Very low	Very low	Very high	Very high	Medium
Medium	Very high	High	Medium	Medium
Very high	Very high	Very low	Very low	Low
Very low	Very high	Very high	Very low	Low
Very high	Very high	Very low	Very high	Low
Very high	Very low	Very high	Very high	Medium
Very low	Very high	Very high	Very high	Medium
Medium	Very low	Medium	Very high	Low
Medium	High	Low	Medium	Low
Very low	Medium	Very high	Medium	Medium

Calculated Parameters

type	wmin
W1	5
W2	5
W3	5
W4	5
var	0.0005

Node: Daily Scrum quality

Truth Table

Monitor sprint progress	All participants present	Answer the 3 basic questions	15 minutes limit	Expected
Very high	Very high	Very low	Medium	Medium
Very low	Very low	Very high	Very high	Low
Medium	Very high	High	Medium	High
Very high	Very low	Very low	Very low	Very low
Very low	Very high	Very low	Very low	Very low
Very low	Very low	Very high	Very low	Low
Very low	Very low	Very low	Very high	Very low
Very high	Very high	Very low	Very low	Low
Very low	Very high	Very high	Very low	<Medium
Very high	Very high	Very high	Very low	High
Very high	Very high	Very low	Very high	Low
Very high	Very low	Very high	Very high	Medium
Very low	Very high	Very high	Very high	High

Calculated Parameters

type	wmin
W1	5
W2	5
W3	5
W4	5
var	0.0005

Node: Product Owner personal characteristics

Truth Table

Communicator and negotiator	Empowered and committed	Available and qualified	Visionary and doer	Leader and team player	Expected
Very high	Very high	Very high	Very high	Very low	Low
Very high	Very high	Very high	Very low	Very high	Low
Very high	Very high	Very low	Very high	Very high	Low
Very high	Very low	Very high	Very high	Very high	Low
Very low	Very low	Very low	Very low	Very high	Low
Very low	Very low	Very low	Very high	Very low	Low
Very low	Very low	Very high	Very low	Very low	Low
Very low	Very high	Very low	Very low	Very low	Low
Very high	Very low	Very low	Very low	Very low	Low

Calculated Parameters

type	wmin
W1	5
W2	5
W3	5
W4	5
W5	5
var	0.0005

Node: Development team teamwork skills

Truth Table

Team Size	Team physical distribution	Collective ownership	Members motivation	Members expertise	Members personality	Expected
Very high	Very high	Very high	Very high	Very high	Very low	Low
Very high	Very high	Very high	Very high	Very low	Very high	Low
Very high	Very high	Very high	Very low	Very high	Very high	Low
Very high	Very high	Very low	Very high	Very high	Very high	Medium
Very high	Very low	Very high	Very high	Very high	Very high	High
Very low	Very high	Very high	Very high	Very high	Very high	Medium
Very low	Very low	Very low	Very low	Very low	Very high	High
Very low	Very low	Very low	Very low	Very high	Very low	Medium
Very low	Very low	Very low	Very high	Very low	Very low	Low
Very low	Very low	Very high	Very low	Very low	Very low	Low
Very low	Very high	Very low	Very low	Very low	Very low	Low
Very high	Very low	Very low	Very low	Very low	Very low	Low

Calculated Parameters

type	wmin
W1	5
W2	5
W3	5
W4	5
W5	5
W6	5
var	0.0005

Node: Product backlog items are properly detailed

Truth Table

Team Size	Team physical distribution	Collective ownership	Members motivation	Members expertise	Members personality	Expected
Very high	Very high	Very high	Very high	Very high	Very low	Low
Very high	Very high	Very high	Very high	Very low	Very high	Low
Very high	Very high	Very high	Very low	Very high	Very high	Low
Very high	Very high	Very low	Very high	Very high	Very high	Medium
Very high	Very low	Very high	Very high	Very high	Very high	High
Very low	Very high	Very high	Very high	Very high	Very high	Medium
Very low	Very low	Very low	Very low	Very low	Very high	High
Very low	Very low	Very low	Very low	Very high	Very low	Medium
Very low	Very low	Very low	Very high	Very low	Very low	Low
Very low	Very low	Very high	Very low	Very low	Very low	Low
Very low	Very high	Very low	Very low	Very low	Very low	Low
Very high	Very low	Very low	Very low	Very low	Very low	Low

Calculated Parameters

type	wmin
W1	5
W2	5
W3	5
W4	5
W5	5
W6	5
var	0.0005

Apêndice C

Código Método Força Bruta

```cpp
#pragma warning(disable:4786)       // disable debug warning

#include <iostream>
#include <vector>                   // for vector class
#include <string>                   // for string class
#include <algorithm>                // for sort algorithm
#include <time.h>                   // for random seed
#include <math.h>                   // for abs()
#include <conio.h>
#include "genomeRankedNode.h"
#include "brierScore.h"

using namespace std;

int main() {

    string fileName;

    // get file name
    fileName = "teste.txt";

    BrierScore* brierScore = new BrierScore(fileName);
    int numParents = brierScore->getNumParents();

    shared_ptr<GenomeRankedNode> best;
    // a huge number...
    double bestBrierScore = 123456789;

    timestamp();
    std::cout << "\n";
    std::cout << "Starting the calculations.\n";

    //loop todos os tipos de funcoes
    for(int k = 0; k < 3; k++){
        //loop todos os tipos de variancias
        for (int v = 0; v < 11; v++){
            // "loop" flexivel de acordo com o numero de nos pai
            int max = 5;
            int depth = numParents;
            // Initialize the slots to hold the current iteration value for each depth (in this case, 1)
            int* slots = (int*)alloca(sizeof(int) * depth);
            for (int i = 0; i < depth; i++){
                slots[i] = 1;
            }

            // index points to the current variable being incremented
            int index = 0;

            while (index < depth){
                // Carry
                if (slots[index] == max)
                    index++;

                shared_ptr<GenomeRankedNode> node = make_shared<GenomeRankedNode>(k, numParents, slots, v);
                cout << node->toString();
                double calcBrierScore = brierScore->calculateBrierScore(node);
                cout << "Brier score: " << calcBrierScore << "\n";

                if (calcBrierScore < bestBrierScore) {
                    best = node;
                    bestBrierScore = calcBrierScore;
                }

                // Increment
                slots[index]++;

            }
        }
    }
    cout << "\nThe best Brier Score: " << bestBrierScore;
    cout << "\nThe best configuration: " << best->toString();

    timestamp();
    getch();
    return 0;
}
```

Figura C.1: main.cpp

```
#include "brierScore.h"

BrierScore::BrierScore(string inFileName)
{
    fileName = inFileName;
    readValuesFromFile();
}

BrierScore::~BrierScore(void)
{
}

int BrierScore::getNumParents() const
{
    return numParents;
}

string BrierScore::getFileName() const
{
    return fileName;
}

void BrierScore::setFileName(string inFileName){
    fileName = inFileName;
}

// Dependent on the number of parents
double BrierScore::calculateBrierScore(shared_ptr<GenomeRankedNode> inNode)
{
    //vector<double> score;
    double meanScore = 0;

    vector<shared_ptr<RNode>> parents;
    GenomeRankedNode node = inNode;
    int inNumParents = inNode->getNumParents();
    int inFunction = inNode->getFunction();
    int *inWeights = inNode->getWeights();
    int inVariance = inNode->getVariance();
    vector<double>* calculatedStates;
    vector<double> brierScore;

    // deep copy of the input data
    vector<double> tempData;
    for (int i = 0; i < inputData.size(); i++)
        tempData.push_back(inputData.at(i));

    // create child node
    shared_ptr<RNode> child = make_shared<RNode>();
    child-> withName("Child")
        ->withVariance(inNode->getRealVariance(inVariance))
        ->withState("very low")
        ->withState("low")
        ->withState("medium")
        ->withState("high")
        ->withState("very high")
        ->init();

    // add parent nodes
    for (int i = 0; i < inNumParents; i++){
        parents.push_back(make_shared<RNode>());
        parents[i]-> withName("Parent")
-           ->withState("very low")
            ->withState("low")
            ->withState("medium")
            ->withState("high")
            ->withState("very high")
            ->init();
        child->addParent(parents[i], inWeights[i]);
    }
```

```
    // Handle first half of cases
    // Set all parents to very low
    setVeryLow(parents);

    for (int i = 0; i < inNumParents; i++){
        // Set a given node to Very High
        parents.at(i)->generateSamples(.9);

        // set the function
        if (inFunction == 0)
            child->wmean();
        else if (inFunction == 1)
            child->wmax();
        else if (inFunction == 2)
            child->wmin();
        else
            child->mixMinMax(inWeights[0], inWeights[1]);

        // collect the given tpn line values (getStateValuePtr)
        calculatedStates = child->getStateValuesPtr();

        double score = 0;
        // calculate the brier score
        for (int j = 0; j < 5; j++){
            // get the first value read from the file
            double currentData = tempData.front();
            // remove the first value, because it will not be necessary anymore
            tempData.erase(tempData.begin());
            score += pow(calculatedStates->at(j) - currentData,2);

        }
        brierScore.push_back(score);

        // set all node samples to Very Low
        setVeryLow(parents);
    }

    // Handle second half of cases
    // Set all parents to very high
    if (numParents > 2) {
        BrierScore::setVeryHigh(parents);
        for (int i = 0; i < inNumParents; i++){
            // Set a given node to Very Low
            parents.at(i)->generateSamples(.1);

            // set the function
            if (inFunction == 0)
                child->wmean();
            else if (inFunction == 1)
                child->wmax();
            else if (inFunction == 2)
                child->wmin();
            else
                child->mixMinMax(inWeights[0], inWeights[1]);

            // collect the given tpn line values (getStateValuePtr)
            calculatedStates = child->getStateValuesPtr();

            double score = 0;
            // calculate the brier score
            for (int j = 0; j < 5; j++){
                // get the first value read from the file
                double currentData = tempData.front();
                // remove the first value, because it will not be necessary anymore
                tempData.erase(tempData.begin());
                score += pow(calculatedStates->at(j) - currentData,2);
                //score.push_back(pow(calculatedStates->at(j) - currentData,2));
            }
            brierScore.push_back(score);
            // set all node samples to Very Low
            setVeryHigh(parents);
        }
    }
```

```
    // if numParents == 2, handle two extra cases
    if (numParents == 2){
        //handle VL M
        parents[0]->generateSamples(.1);
        parents[1]->generateSamples(.5);

        // set the function
        if (inFunction == 0)
            child->wmean();
        else if (inFunction == 1)
            child->wmax();
        else if (inFunction == 2)
            child->wmin();
        else
            child->mixMinMax(inWeights[0], inWeights[1]);

        calculatedStates = child->getStateValuesPtr();

        double score = 0;
        // calculate the brier score
        for (int j = 0; j < 5; j++){
            // get the first value read from the file
            double currentData = tempData.front();

            // remove the first value, because it will not be necessary anymore
            tempData.erase(tempData.begin());
            score += pow(calculatedStates->at(j) - currentData,2);
        }
        brierScore.push_back(score);

        //handle M VL
        parents[0]->generateSamples(.5);
        parents[1]->generateSamples(.1);

        // set the function
        if (inFunction == 0)
            child->wmean();
        else if (inFunction == 1)
            child->wmax();
        else if (inFunction == 2)
            child->wmin();
        else
            child->mixMinMax(inWeights[0], inWeights[1]);

        calculatedStates = child->getStateValuesPtr();

        score = 0;
        // calculate the brier score
        for (int j = 0; j < 5; j++){
            // get the first value read from the file
            double currentData = tempData.front();

            // remove the first value, because it will not be necessary anymore
            tempData.erase(tempData.begin());
            score += pow(calculatedStates->at(j) - currentData,2);
        }
        brierScore.push_back(score);
    }

    double totalScore = 0;
    for (int i = 0; i < (numParents*2); i++)
        totalScore +=brierScore[i];

    meanScore = totalScore / (numParents*2);

    return meanScore;
}
```

```
// reads all values from the file and store them in an array
void BrierScore::readValuesFromFile()
{
    std::ifstream infile(fileName);

    if( !infile  ) {
        std::cerr << "Can't open file " << fileName << std::endl;
        std::exit( -1 );
    }

    double currentValue;
    infile >> numParents;

    // Get the expected data from the file
    // numNos*2 - number of questions
    // 5 - number of states
    //inputData = new double[numParents*2*5];

    for (int i = 0; i < (numParents*2); i++){
        for (int j = 0; j < 5; j++){
            infile >> currentValue;
            inputData.push_back(currentValue);
            //infile >> inputData[j+i*5];
        }
    }
}

void BrierScore::setVeryLow(vector<std::shared_ptr<RNode>> parents){
    for (int i = 0; i < parents.size(); i++)
        parents.at(i)->generateSamples(.1);
}

void BrierScore::setVeryHigh(vector<std::shared_ptr<RNode>> parents){
    for (int i = 0; i < parents.size(); i++)
        parents.at(i)->generateSamples(.9);
}
```

Figura C.2: brierScore.cpp

```
#define _CRT_SECURE_NO_WARNINGS
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <cmath>
# include <ctime>
# include <cstring>

using namespace std;

# include "truncated_normal.hpp"

//****************************************************************************80

double normal_01_cdf ( double x )

//****************************************************************************80
//
//  Purpose:
//
//    NORMAL_01_CDF evaluates the Normal 01 CDF.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    10 February 1999
//
//  Author:
//
//    John Burkardt
//
//  Reference:
//
//    A G Adams,
//    Areas Under the Normal Curve,
//    Algorithm 39,
//    Computer j.,
//    Volume 12, pages 197-198, 1969.
//
//  Parameters:
//
//    Input, double X, the argument of the CDF.
//
//    Output, double CDF, the value of the CDF.
//
{
  double a1 = 0.398942280444;
  double a2 = 0.399903438504;
  double a3 = 5.75885480458;
  double a4 = 29.8213557808;
  double a5 = 2.62433121679;
  double a6 = 48.6959930692;
  double a7 = 5.92885724438;
  double b0 = 0.398942280385;
  double b1 = 3.8052E-08;
  double b2 = 1.00000615302;
  double b3 = 3.98064794E-04;
  double b4 = 1.98615381364;
  double b5 = 0.151679116635;
  double b6 = 5.29330324926;
  double b7 = 4.8385912808;
  double b8 = 15.1508972451;
  double b9 = 0.742380924027;
  double b10 = 30.789933034;
  double b11 = 3.99019417011;
  double cdf;
  double q;
  double y;
```

```
//  |X| <= 1.28.
//
  if ( fabs ( x ) <= 1.28 )
  {
    y = 0.5 * x * x;

    q = 0.5 - fabs ( x ) * ( a1 - a2 * y / ( y + a3 - a4 / ( y + a5
      + a6 / ( y + a7 ) ) ) );
//
//  1.28 < |X| <= 12.7
//
  }
  else if ( fabs ( x ) <= 12.7 )
  {
    y = 0.5 * x * x;

    q = exp ( - y ) * b0 / ( fabs ( x ) - b1
      + b2 / ( fabs ( x ) + b3
      + b4 / ( fabs ( x ) - b5
      + b6 / ( fabs ( x ) + b7
      - b8 / ( fabs ( x ) + b9
      + b10 / ( fabs ( x ) + b11 ) ) ) ) ) );
//
//  12.7 < |X|
//
  }
  else
  {
    q = 0.0;
  }
//
//  Take account of negative X.
//
  if ( x < 0.0 )
  {
    cdf = q;
  }
  else
  {
    cdf = 1.0 - q;
  }

  return cdf;
}
//****************************************************************************80

double normal_01_cdf_inv ( double p )

//****************************************************************************80
//
//  Purpose:
//
//    NORMAL_01_CDF_INV inverts the standard normal CDF.
//
//  Discussion:
//
//    The result is accurate to about 1 part in 10**16.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    27 December 2004
//
//  Author:
//
//    Original FORTRAN77 version by Michael Wichura.
//    C++ version by John Burkardt.
//
```

```
//  Reference:
//
//    Michael Wichura,
//    The Percentage Points of the Normal Distribution,
//    Algorithm AS 241,
//    Applied Statistics,
//    Volume 37, Number 3, pages 477-484, 1988.
//
//  Parameters:
//
//    Input, double P, the value of the cumulative probability
//    densitity function.  0 < P < 1.  If P is outside this range, an
//    "infinite" value is returned.
//
//    Output, double NORMAL_01_CDF_INV, the normal deviate value
//    with the property that the probability of a standard normal deviate being
//    less than or equal to this value is P.
//
{
  double a[8] = {
    3.3871328727963666080,     1.3314166789178437745e+2,
    1.9715909503065514427e+3,  1.3731693765509461125e+4,
    4.5921953931549871457e+4,  6.7265770927008700853e+4,
    3.3430575583588128105e+4,  2.5090809287301226727e+3 };
  double b[8] = {
    1.0,                       4.2313330701600911252e+1,
    6.8718700749205790830e+2,  5.3941960214247511077e+3,
    2.1213794301586595867e+4,  3.9307895800092710610e+4,
    2.8729085735721942674e+4,  5.2264952788528545610e+3 };
  double c[8] = {
    1.42343711074968357734,     4.63033784615654529590,
    5.76949722146069140550,     3.64784832476320460504,
    1.27045825245236838258,     2.41780725177450611770e-1,
    2.27238449892691845833e-2,  7.74545014278341407640e-4 };
  double const1 = 0.180625;
  double const2 = 1.6;
  double d[8] = {
    1.0,                        2.05319162663775882187,
    1.67638483018380384940,     6.89767334985100004550e-1,
    1.48103976427480074590e-1,  1.51986665636164571966e-2,
    5.47593808499534494600e-4,  1.05075007164441684324e-9 };
  double e[8] = {
    6.65790464350110377720,     5.46378491116411436990,
    1.78482653991729133580,     2.96560571828504891230e-1,
    2.65321895265761230930e-2,  1.24266094738807843860e-3,
    2.71155556874348757815e-5,  2.01033439929228813265e-7 };
  double f[8] = {
    1.0,                        5.99832206555887937690e-1,
    1.36929880922735805310e-1,  1.48753612908506148525e-2,
    7.86869131145613259100e-4,  1.84631831751005468180e-5,
    1.42151175831644588870e-7,  2.04426310338993978564e-15 };
  double q;
  double r;
  double split1 = 0.425;
  double split2 = 5.0;
  double value;

  if ( p <= 0.0 )
  {
    value = -r8_huge ( );
    return value;
  }

  if ( 1.0 <= p )
  {
    value = r8_huge ( );
    return value;
  }

  q = p - 0.5;
```

```
  if ( fabs ( q ) <= split1 )
  {
    r = const1 - q * q;
    value = q * r8poly_value ( 8, a, r ) / r8poly_value ( 8, b, r );
  }
  else
  {
    if ( q < 0.0 )
    {
      r = p;
    }
    else
    {
      r = 1.0 - p;
    }

    if ( r <= 0.0 )
    {
      value = r8_huge ( );
    }
    else
    {
      r = sqrt ( - log ( r ) );

      if ( r <= split2 )
      {
        r = r - const2;
        value = r8poly_value ( 8, c, r ) / r8poly_value ( 8, d, r );
       }
       else
       {
         r = r - split2;
         value = r8poly_value ( 8, e, r ) / r8poly_value ( 8, f, r );
       }
    }

    if ( q < 0.0 )
    {
      value = - value;
    }
  return value;
}
//****************************************************************************80

double normal_01_mean ( )

//****************************************************************************80
//
//  Purpose:
//
//    NORMAL_01_MEAN returns the mean of the Normal 01 PDF.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    18 September 2004
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Output, double MEAN, the mean of the PDF.
//
{
  double mean;

  mean = 0.0;

  return mean;
}
```

```
double normal_01_moment ( int order )

//****************************************************************************80
//
//  Purpose:
//
//    NORMAL_01_MOMENT evaluates moments of the Normal 01 PDF.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    31 August 2013
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Input, int ORDER, the order of the moment.
//    0 <= ORDER.
//
//    Output, double NORMAL_01_MOMENT, the value of the moment.
//
{
  double value;

  if ( ( order % 2 ) == 0 )
  {
    value = r8_factorial2 ( order - 1 );
  }
  else
  {
    value = 0.0;
  }
  return value;
}
//****************************************************************************80

double normal_01_pdf ( double x )

//****************************************************************************80
//
//  Purpose:
//    NORMAL_01_PDF evaluates the Normal 01 PDF.
//
//  Discussion:
//    The Normal 01 PDF is also called the "Standard Normal" PDF, or
//    the Normal PDF with 0 mean and variance 1.
//
//    PDF(X) = exp ( - 0.5 * X^2 ) / sqrt ( 2 * PI )
//
//  Licensing:
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//    18 September 2004
//
//  Author:
//    John Burkardt
//
//  Parameters:
//    Input, double X, the argument of the PDF.
//    Output, double PDF, the value of the PDF.
//
{
  double pdf;
  const double pi = 3.14159265358979323;

  pdf = exp ( -0.5 * x * x ) / sqrt ( 2.0 * pi );

  return pdf;
}
```

```
double normal_01_sample ( int &seed )

//****************************************************************************80
//
//  Purpose:
//    NORMAL_01_SAMPLE samples the standard normal probability distribution.
//
//  Discussion:
//
//    The standard normal probability distribution function (PDF) has
//    mean 0 and standard deviation 1.
//
//    The Box-Muller method is used, which is efficient, but
//    generates two values at a time.
//
//  Licensing:
//    This code is distributed under the GNU LGPL license.
//  Author:
//
//    John Burkardt
//
//  Parameters:
//    Input/output, int &SEED, a seed for the random number generator.
//    Output, double NORMAL_01_SAMPLE, a normally distributed random value.
//
{
  const double pi = 3.14159265358979323;
  double r1;
  double r2;
  double x;

  r1 = r8_uniform_01 ( seed );
  r2 = r8_uniform_01 ( seed );
  x = sqrt ( -2.0 * log ( r1 ) ) * cos ( 2.0 * pi * r2 );

  return x;
}
//****************************************************************************80
double normal_01_variance ( )

//****************************************************************************80
//
//  Purpose:
//    NORMAL_01_VARIANCE returns the variance of the Normal 01 PDF.
//
//  Licensing:
//    This code is distributed under the GNU LGPL license.
//
//  Author:
//    John Burkardt
//
//  Parameters:
//    Output, double VARIANCE, the variance of the PDF.
//
{
  double variance;

  variance = 1.0;

  return variance;
}
//****************************************************************************80
```

```
double normal_cdf ( double x, double a, double b )

//****************************************************************************80
//
//  Purpose:
//    NORMAL_CDF evaluates the Normal CDF.
//
//  Licensing:
//    This code is distributed under the GNU LGPL license.
//
//  Author:
//    John Burkardt
//
//  Parameters:
//    Input, double X, the argument of the CDF.
//    Input, double A, B, the parameters of the PDF.
//    0.0 < B.
//    Output, double CDF, the value of the CDF.
//
{
  double cdf;
  double y;

  y = ( x - a ) / b;

  cdf = normal_01_cdf ( y );

  return cdf;
}
double normal_cdf_inv ( double cdf, double a, double b )
//  Purpose:
//    NORMAL_CDF_INV inverts the Normal CDF.
//
//  Licensing:
//    This code is distributed under the GNU LGPL license.
//
//  Author:
//    John Burkardt
//
//  Reference:
//    Algorithm AS 111,
//    Applied Statistics,
//    Volume 26, pages 118-121, 1977.
//
//  Parameters:
//    Input, double CDF, the value of the CDF.
//    0.0 <= CDF <= 1.0.
//    Input, double A, B, the parameters of the PDF.
//    0.0 < B.
//    Output, double NORMAL_CDF_INV, the corresponding argument.
{
  double x;
  double x2;

  if ( cdf < 0.0 || 1.0 < cdf )
  {
    cout << "\n";
    cout << "NORMAL_CDF_INV - Fatal error!\n";
    cout << "  CDF < 0 or 1 < CDF.\n";
    exit ( 1 );
  }

  x2 = normal_01_cdf_inv ( cdf );

  x = a + b * x2;

  return x;
}
```

```
double normal_mean ( double a, double b )
//  Purpose:
//    NORMAL_MEAN returns the mean of the Normal PDF.
//
//  Licensing:
//    This code is distributed under the GNU LGPL license.
//
//  Author:
//    John Burkardt
//
//  Parameters:
//    Input, double A, B, the parameters of the PDF.
//    0.0 < B.
//    Output, double MEAN, the mean of the PDF.
{
  double mean;

  mean = a;

  return mean;
}
double normal_moment ( int order, double mu, double sigma )
//  Purpose:
//    NORMAL_MOMENT evaluates moments of the Normal PDF.
//
//  Discussion:
//    The formula was posted by John D Cook.
//
//    Order  Moment
//    -----  ------
//      0    1
//      1    mu
//      2    mu^2 +         sigma^2
//      3    mu^3 +  3 mu   sigma^2
//      4    mu^4 +  6 mu^2 sigma^2 +   3      sigma^4
//      5    mu^5 + 10 mu^3 sigma^2 +  15 mu   sigma^4
//      6    mu^6 + 15 mu^4 sigma^2 +  45 mu^2 sigma^4 +  15      sigma^6
//      7    mu^7 + 21 mu^5 sigma^2 + 105 mu^3 sigma^4 + 105 mu   sigma^6
//      8    mu^8 + 28 mu^6 sigma^2 + 210 mu^4 sigma^4 + 420 mu^2 sigma^6
//           + 105 sigma^8
//
//  Licensing:
//    This code is distributed under the GNU LGPL license.
//
//  Author:
//    John Burkardt
//
//  Parameters:
//    Input, int ORDER, the order of the moment.
//    0 <= ORDER.
//    Input, double MU, the mean of the distribution.
//    Input, double SIGMA, the standard deviation of the distribution.
//    Output, double NORMAL_MOMENT, the value of the central moment.
{
  int j;
  int j_hi;
  double value;

  j_hi = ( order / 2 );

  value = 0.0;
  for ( j = 0; j <= j_hi; j++ )
  {
    value = value
      + r8_choose ( order, 2 * j )
      * r8_factorial2 ( 2 * j - 1 )
      * pow ( mu, order - 2 * j ) * pow ( sigma, 2 * j );
  }

  return value;
}
```

```
double normal_moment_central ( int order, double mu, double sigma )
//  Purpose:
//    NORMAL_MOMENT_CENTRAL evaluates central moments of the Normal PDF.
{
  double value;

  if ( ( order % 2 ) == 0 )
  {
    value = r8_factorial2 ( order - 1 ) * pow ( sigma, order );
  }
  else
  {
    value = 0.0;
  }

  return value;
}
double normal_moment_central_values ( int order, double mu, double sigma )
//  Purpose:
//    NORMAL_MOMENT_CENTRAL_VALUES: moments 0 through 10 of the Normal PDF.
{
  double value;

  if ( order == 0 )
  {
    value = 1.0;
  }
  else if ( order == 1 )
  {
    value = 0.0;
  }
  else if ( order == 2 )
  {
    value = pow ( sigma, 2 );
  }
  else if ( order == 3 )
  {
    value = 0.0;
  }
  else if ( order == 4 )
  {
    value = 3.0 * pow ( sigma, 4 );
  }
  else if ( order == 5 )
  {
    value = 0.0;
  }
  else if ( order == 6 )
  {
    value = 15.0 * pow ( sigma, 6 );
  }
  else if ( order == 7 )
  {
    value = 0.0;
  }
  else if ( order == 8 )
  {
    value = 105.0 * pow ( sigma, 8 );
  }
  else if ( order == 9 )
  {
    value = 0.0;
```

```
double normal_moment_values ( int order, double mu, double sigma )
//  Purpose:
//    NORMAL_MOMENT_VALUES evaluates moments 0 through 8 of the Normal PDF.
{
  double value;

  if ( order == 0 )
  {
    value = 1.0;
  }
  else if ( order == 1 )
  {
    value = mu;
  }
  else if ( order == 2 )
  {
    value = pow ( mu, 2 ) + pow ( sigma, 2 );
  }
  else if ( order == 3 )
  {
    value = pow ( mu, 3 ) + 3.0 * mu * pow ( sigma, 2 );
  }
  else if ( order == 4 )
  {
    value = pow ( mu, 4 ) + 6.0 * pow ( mu, 2 ) * pow ( sigma, 2 )
      + 3.0 * pow ( sigma, 4 );
  }
  else if ( order == 5 )
  {
    value = pow ( mu, 5 ) + 10.0 * pow ( mu, 3 ) * pow ( sigma, 2 )
      + 15.0 * mu * pow ( sigma, 4 );
  }
  else if ( order == 6 )
  {
    value = pow ( mu, 6 ) + 15.0 * pow ( mu, 4 ) * pow ( sigma, 2 )
      + 45.0 * pow ( mu, 2 ) * pow ( sigma, 4 )
      + 15.0 * pow ( sigma, 6 );
  }
  else if ( order == 7 )
  {
    value = pow ( mu, 7 ) + 21.0 * pow ( mu, 5 ) * pow ( sigma, 2 )
      + 105.0 * pow ( mu, 3 ) * pow ( sigma, 4 )
      + 105.0 * mu * pow ( sigma, 6 );
  }
  else if ( order == 8 )
  {
    value = pow ( mu, 8 ) + 28.0 * pow ( mu, 6 ) * pow ( sigma, 2 )
      + 210.0 * pow ( mu, 4 ) * pow ( sigma, 4 )
      + 420.0 * pow ( mu, 2 ) * pow ( sigma, 6 ) + 105.0 * pow ( sigma, 8 );
  }
  else
  {
    cerr << "\n";
    cerr << "NORMAL_MOMENT_VALUES - Fatal error!\n";
    cerr << "  Only ORDERS 0 through 8 are available.\n";
    exit ( 1 );
  }

  return value;
```

```
double normal_pdf ( double x, double a, double b )
//  Purpose:
//    NORMAL_PDF evaluates the Normal PDF.
{
  double pdf;
  const double pi = 3.14159265358979323;
  double y;

  y = ( x - a ) / b;

  pdf = exp ( -0.5 * y * y )  / ( b * sqrt ( 2.0 * pi ) );

  return pdf;
}

double normal_sample ( double a, double b, int &seed )
//  Purpose:
//    NORMAL_SAMPLE samples the Normal PDF.
{
  double x;
  x = normal_01_sample ( seed );
  x = a + b * x;

  return x;
}

double normal_variance ( double a, double b )
//  Purpose:
//    NORMAL_VARIANCE returns the variance of the Normal PDF.
{
  double variance;
  variance = b * b;
  return variance;
}

double r8_abs ( double x )
//  Purpose:
//    R8_ABS returns the absolute value of an R8.
{
  double value;

  if ( 0.0 <= x ) {
    value = x;
  } else {
    value = -x;
  }
  return value;
}

double r8_choose ( int n, int k )
//  Purpose:
//    R8_CHOOSE computes the binomial coefficient C(N,K) as an R8.
{
  int i;
  int mn;
  int mx;
  double value;

  if ( k < n - k ) {
    mn = k;
    mx = n - k;
  } else {
    mn = n - k;
    mx = k;
  }

  if ( mn < 0 ) {
    value = 0.0;
  } else if ( mn == 0 ) {
    value = 1.0;
  } else {
    value = ( double ) ( mx + 1 );
    for ( i = 2; i <= mn; i++ ) {
      value = ( value * ( double ) ( mx + i ) ) / ( double ) i;
    }
  }

  return value;
```

```
double r8_factorial2 ( int n )
//  Purpose:
//    R8_FACTORIAL2 computes the double factorial function.
{
  int n_copy;
  double value;

  value = 1.0;

  if ( n < 1 ) {
    return value;
  }

  n_copy = n;
  while ( 1 < n_copy ) {
    value = value * ( double ) n_copy;
    n_copy = n_copy - 2;
  }

  return value;
}

double r8_huge ( )
//  Purpose:
//    R8_HUGE returns a "huge" R8.
{
  return HUGE_VAL;
}

double r8_log_2 ( double x )
//  Purpose:
//    R8_LOG_2 returns the logarithm base 2 of the absolute value of an R8.
{
  double value;

  if ( x == 0.0 ) {
    value = - r8_huge ( );
  } else {
    value = log ( fabs ( x ) ) / log ( 2.0 );
  }

  return value;
}

double r8_mop ( int i )
//  Purpose:
//    R8_MOP returns the I-th power of -1 as an R8 value.
{
  double value;
  if ( ( i % 2 ) == 0 ) {
    value = 1.0;
  } else {
    value = -1.0;
  }
  return value;
}

double r8_uniform_01 ( int &seed )
//  Purpose:
//    R8_UNIFORM_01 returns a unit pseudorandom R8.
{
  int k;
  double r;

  k = seed / 127773;
  seed = 16807 * ( seed - k * 127773 ) - k * 2836;
  if ( seed < 0 ) {
    seed = seed + 2147483647;
  }
  r = ( double ) ( seed ) * 4.656612875E-10;

  return r;
}
```

```c
double r8poly_value ( int n, double a[], double x )
//  Purpose:
//    R8POLY_VALUE evaluates a double precision polynomial.
{
  int i;
  double value;
  value = 0.0;
  for ( i = n-1; 0 <= i; i-- ) {
    value = value * x + a[i];
  }

  return value;
}

double r8vec_max ( int n, double *dvec )
//  Purpose:
//    R8VEC_MAX returns the value of the maximum element in an R8VEC.
{
  int i;
  double rmax;
  double *r8vec_pointer;

  if ( n <= 0 ) {
    return 0.0;
  }

  r8vec_pointer = dvec;
  for ( i = 0; i < n; i++ ) {
    if ( i == 0 ) {
      rmax = *dvec;
      r8vec_pointer = dvec;
    } else {
      r8vec_pointer++;
      if ( rmax < *r8vec_pointer ) {
        rmax = *r8vec_pointer;
      }
    }
  }
  return rmax;
}

double r8vec_mean ( int n, double x[] )
//  Purpose:
//    R8VEC_MEAN returns the mean of an R8VEC.
{
  int i;
  double mean;
  mean = 0.0;
  for ( i = 0; i < n; i++ ) {
    mean = mean + x[i];
  }
  mean = mean / ( double ) n;

  return mean;
}

double r8vec_min ( int n, double *dvec )
//  Purpose:
//    R8VEC_MIN returns the value of the minimum element in an R8VEC.
{
  int i;
  double rmin;
  double *r8vec_pointer;

  if ( n <= 0 ) return 0.0;

  r8vec_pointer = dvec;
  for ( i = 0; i < n; i++ ) {
    if ( i == 0 ) {
      rmin = *dvec;
      r8vec_pointer = dvec;
    } else {
      r8vec_pointer++;
      if ( *r8vec_pointer < rmin ) rmin = *r8vec_pointer;
    }
  }
  return rmin;
```

```
double r8vec_variance ( int n, double x[] )
//  Purpose:
//    R8VEC_VARIANCE returns the variance of an R8VEC.
{
  int i;
  double mean;
  double variance;

  mean = r8vec_mean ( n, x );
  variance = 0.0;
  for ( i = 0; i < n; i++ )
    variance = variance + ( x[i] - mean ) * ( x[i] - mean );

  if ( 1 < n ) {
    variance = variance / ( double ) ( n - 1 );
  } else {
    variance = 0.0;
  }

  return variance;
}

void timestamp ( )
//  Purpose:
//    TIMESTAMP prints the current YMDHMS date as a time stamp.
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct std::tm *tm_ptr;
  size_t len;
  std::time_t now;

  now = std::time ( NULL );
  tm_ptr = std::localtime ( &now );

  len = std::strftime ( time buffer, TIME SIZE, "%d %B %Y %I:%M:%S %p", tm ptr );
  std::cout << time_buffer << "\n";

  return;
# undef TIME_SIZE
}

double truncated_normal_ab_cdf ( double x, double mu, double s, double a, double b )
//  Purpose:
//    TRUNCATED_NORMAL_AB_CDF evaluates the truncated Normal CDF.
{
  double alpha;
  double alpha_cdf;
  double beta;
  double beta_cdf;
  double cdf;
  double xi;
  double xi_cdf;

  alpha = ( a - mu ) / s;
  beta = ( b - mu ) / s;
  xi = ( x - mu ) / s;

  alpha_cdf = normal_01_cdf ( alpha );
  beta_cdf = normal_01_cdf ( beta );
  xi_cdf = normal_01_cdf ( xi );

  cdf = ( xi_cdf - alpha_cdf ) / ( beta_cdf - alpha_cdf );

  return cdf;
}
```

```
double truncated_normal_ab_cdf ( double x, double mu, double s, double a, double b )
//  Purpose:
//    TRUNCATED_NORMAL_AB_CDF evaluates the truncated Normal CDF.
{
  double alpha;
  double alpha_cdf;
  double beta;
  double beta_cdf;
  double cdf;
  double xi;
  double xi_cdf;

  alpha = ( a - mu ) / s;
  beta = ( b - mu ) / s;
  xi = ( x - mu ) / s;

  alpha_cdf = normal_01_cdf ( alpha );
  beta_cdf = normal_01_cdf ( beta );
  xi_cdf = normal_01_cdf ( xi );

  cdf = ( xi_cdf - alpha_cdf ) / ( beta_cdf - alpha_cdf );

  return cdf;
}

void truncated_normal_ab_cdf_values ( int &n_data, double &mu, double &sigma,
  double &a, double &b, double &x, double &fx )
//  Purpose:
//    TRUNCATED_NORMAL_AB_CDF_VALUES: values of the Truncated Normal CDF.
{
# define N_MAX 11

  static double a_vec[N_MAX] = {50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0,
     50.0, 50.0, 50.0, 50.0 };

  static double b_vec[N_MAX] = {150.0, 150.0, 150.0, 150.0, 150.0,
     150.0, 150.0, 150.0, 150.0, 150.0, 150.0 };
  static double fx_vec[N_MAX] = {0.3371694242213513, 0.3685009225506048, 0.4006444233448185,
    0.4334107066903040, 0.4665988676496338, 0.5000000000000000, 0.5334011323503662,
    0.5665892933096960, 0.5993555766551815, 0.6314990774493952, 0.6628305757786487 };

  static double mu_vec[N_MAX] = {100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,
      100.0, 100.0, 100.0, 100.0 };

  static double sigma_vec[N_MAX] = {25.0, 25.0, 25.0, 25.0, 25.0, 25.0, 25.0, 25.0,
      25.0, 25.0, 25.0  };

  static double x_vec[N_MAX] = {90.0, 92.0, 94.0, 96.0, 98.0, 100.0,
    102.0, 104.0, 106.0, 108.0, 110.0 };

  if ( n_data < 0 ) {
    n_data = 0;
  }

  n_data = n_data + 1;

  if ( N_MAX < n_data ) {
    n_data = 0;
    a = 0.0;
    b = 0.0;
    mu = 0.0;
    sigma = 0.0;
    x = 0.0;
    fx = 0.0;
  } else {
    a = a_vec[n_data-1];
    b = b_vec[n_data-1];
    mu = mu_vec[n_data-1];
    sigma = sigma_vec[n_data-1];
    x = x_vec[n_data-1];
    fx = fx_vec[n_data-1];
  }

  return;
# undef N_MAX
}
```

```
double truncated_normal_ab_cdf_inv ( double cdf, double mu, double s, double a, double b )
//  Purpose:
//    TRUNCATED_NORMAL_AB_CDF_INV inverts the truncated Normal CDF.
{
  double alpha;
  double alpha_cdf;
  double beta;
  double beta_cdf;
  double x;
  double xi;
  double xi_cdf;

  if ( cdf < 0.0 || 1.0 < cdf ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_AB_CDF_INV - Fatal error!\n";
    cerr << "  CDF < 0 or 1 < CDF.\n";
    exit ( 1 );
  }

  alpha = ( a - mu ) / s;
  beta = ( b - mu ) / s;
  alpha_cdf = normal_01_cdf ( alpha );
  beta_cdf = normal_01_cdf ( beta );
  xi_cdf = ( beta_cdf - alpha_cdf ) * cdf + alpha_cdf;
  xi = normal_01_cdf_inv ( xi_cdf );
  x = mu + s * xi;

  return x;
}
double truncated_normal_ab_mean ( double mu, double s, double a, double b )
//  Purpose:
//    TRUNCATED_NORMAL_AB_MEAN returns the mean of the truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double alpha_pdf;
  double beta;
  double beta_cdf;
  double beta_pdf;
  double mean;

  alpha = ( a - mu ) / s;
  beta = ( b - mu ) / s;
  alpha_cdf = normal_01_cdf ( alpha );
  beta_cdf = normal_01_cdf ( beta  );
  alpha_pdf = normal_01_pdf ( alpha );
  beta_pdf = normal_01_pdf ( beta );
  mean = mu + s * ( alpha_pdf - beta_pdf ) / ( beta_cdf - alpha_cdf );

  return mean;
}

double truncated_normal_ab_moment ( int order, double mu, double s, double a, double b )
//  Purpose:
//    TRUNCATED_NORMAL_AB_MOMENT: moments of the truncated Normal PDF.
{
  double a_cdf;
  double a_h;
  double a_pdf;
  double b_cdf;
  double b_h;
  double b_pdf;
  double ir;
  double irm1;
  double irm2;
  double moment;
  int r;

  if ( order < 0 )
```

```
  if ( order < 0 ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_AB_MOMENT - Fatal error!\n";
    cerr << "  ORDER < 0.\n";
    exit ( 1 );
  }

  if ( s <= 0.0 ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_AB_MOMENT - Fatal error!\n";
    cerr << "  S <= 0.0.\n";
    exit ( 1 );
  }

  if ( b <= a ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_AB_MOMENT - Fatal error!\n";
    cerr << "  B <= A.\n";
    exit ( 1 );
  }
  a_h = ( a - mu ) / s;
  a_pdf = normal_01_pdf ( a_h );
  a_cdf = normal_01_cdf ( a_h );

  if ( a_cdf == 0.0 ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_AB_MOMENT - Fatal error!\n";
    cerr << "  PDF/CDF ratio fails, because A_CDF too small.\n";
    cerr << "  A_PDF = " << a_pdf << "\n";
    cerr << "  A_CDF = " << a_cdf << "\n";
    exit ( 1 );
  }
  b_h = ( b - mu ) / s;
  b_pdf = normal_01_pdf ( b_h );
  b_cdf = normal_01_cdf ( b_h );

  if ( b_cdf == 0.0 ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_AB_MOMENT - Fatal error!\n";
    cerr << "  PDF/CDF ratio fails, because B_CDF too small.\n";
    cerr << "  B_PDF = " << b_pdf << "\n";
    cerr << "  B_CDF = " << b_cdf << "\n";
    exit ( 1 );
  }

  moment = 0.0;
  irm2 = 0.0;
  irm1 = 0.0;

  for ( r = 0; r <= order; r++ ) {
    if ( r == 0 ) {
      ir = 1.0;
    } else if ( r == 1 ) {
      ir = - ( b_pdf - a_pdf ) / ( b_cdf - a_cdf );
    } else {
      ir = ( double ) ( r - 1 ) * irm2
        - ( pow ( b_h, r - 1 ) * b_pdf - pow ( a_h, r - 1 ) * a_pdf )
        / ( b_cdf - a_cdf );
    }

    moment = moment + r8_choose ( order, r ) * pow ( mu, order - r )
      * pow ( s, r ) * ir;

    irm2 = irm1;
    irm1 = ir;
  }
  return moment;
}
```

```
double truncated_normal_ab_pdf ( double x, double mu, double s, double a,  double b )
//  Purpose:
//    TRUNCATED_NORMAL_AB_PDF evaluates the truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double beta;
  double beta_cdf;
  double pdf;
  double xi;
  double xi_pdf;

  alpha = ( a - mu ) / s;
  beta = ( b - mu ) / s;
  xi = ( x - mu ) / s;

  alpha_cdf = normal_01_cdf ( alpha );
  beta_cdf = normal_01_cdf ( beta );
  xi_pdf = normal_01_pdf ( xi );

  pdf = xi_pdf / ( beta_cdf - alpha_cdf ) / s;

  return pdf;
}

void truncated_normal_ab_pdf_values ( int &n_data, double &mu, double &sigma,
  double &a, double &b, double &x, double &fx )
//  Purpose:
//    TRUNCATED_NORMAL_AB_PDF_VALUES: values of the Truncated Normal PDF.
{
# define N_MAX 11
  static double a_vec[N_MAX] = {50.0, 50.0, 50.0, 50.0, 50.0,
     50.0, 50.0, 50.0, 50.0, 50.0, 50.0 };
  static double b_vec[N_MAX] = {150.0, 150.0, 150.0, 150.0, 150.0, 150.0, 150.0,
     150.0, 150.0, 150.0, 150.0 };
  static double fx_vec[N_MAX] = {0.01543301171801836, 0.01588394472270638, 0.01624375997031919,
    0.01650575046469259, 0.01666496869385951, 0.01671838200940538, 0.01666496869385951,
    0.01650575046469259, 0.01624375997031919, 0.01588394472270638, 0.01543301171801836 };
  static double mu_vec[N_MAX] = {100.0, 100.0, 100.0, 100.0, 100.0, 100.0,
     100.0, 100.0, 100.0, 100.0, 100.0 };
  static double sigma_vec[N_MAX] = {25.0, 25.0, 25.0, 25.0, 25.0, 25.0, 25.0,
    25.0, 25.0, 25.0, 25.0  };
  static double x_vec[N_MAX] = {90.0, 92.0, 94.0, 96.0, 98.0, 100.0, 102.0,
    104.0, 106.0, 108.0, 110.0 };

  if ( n_data < 0 )
    n_data = 0;

  n_data = n_data + 1;
  if ( N_MAX < n_data ) {
    n_data = 0;
    a = 0.0;
    b = 0.0;
    mu = 0.0;
    sigma = 0.0;
    x = 0.0;
    fx = 0.0;
  } else {
    a = a_vec[n_data-1];
    b = b_vec[n_data-1];
    mu = mu_vec[n_data-1];
    sigma = sigma_vec[n_data-1];
    x = x_vec[n_data-1];
    fx = fx_vec[n_data-1];
  }
  return;
# undef N_MAX
}
```

```
double truncated_normal_ab_sample ( double mu, double s, double a, double b,  int &seed )
//  Purpose:
//    TRUNCATED_NORMAL_AB_SAMPLE samples the truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double beta;
  double beta_cdf;
  double u;
  double x;
  double xi;
  double xi_cdf;

  alpha = ( a - mu ) / s;
  beta = ( b - mu ) / s;
  alpha_cdf = normal_01_cdf ( alpha );
  beta_cdf = normal_01_cdf ( beta );
  u = r8_uniform_01 ( seed );
  xi_cdf = alpha_cdf + u * ( beta_cdf - alpha_cdf );
  xi = normal_01_cdf_inv ( xi_cdf );
  x = mu + s * xi;

  return x;
}

double truncated_normal_ab_variance ( double mu, double s, double a, double b )
//  Purpose:
//    TRUNCATED_NORMAL_AB_VARIANCE returns the variance of the truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double alpha_pdf;
  double beta;
  double beta_cdf;
  double beta_pdf;
  double variance;

  alpha = ( a - mu ) / s;
  beta = ( b - mu ) / s;
  alpha_pdf = normal_01_pdf ( alpha );
  beta_pdf = normal_01_pdf ( beta );
  alpha_cdf = normal_01_cdf ( alpha );
  beta_cdf = normal_01_cdf ( beta );
  variance = s * s * ( 1.0
    + ( alpha * alpha_pdf - beta * beta_pdf ) / ( beta_cdf - alpha_cdf )
    - pow ( ( alpha_pdf - beta_pdf ) / ( beta_cdf - alpha_cdf ), 2 ) );

  return variance;
}

double truncated_normal_a_cdf ( double x, double mu, double s, double a )
//  Purpose:
//    TRUNCATED_NORMAL_A_CDF evaluates the lower truncated Normal CDF.
{
  double alpha;
  double alpha_cdf;
  double cdf;
  double xi;
  double xi_cdf;

  alpha = ( a - mu ) / s;
  xi = ( x - mu ) / s;

  alpha_cdf = normal_01_cdf ( alpha );
  xi_cdf = normal_01_cdf ( xi );

  cdf = ( xi_cdf - alpha_cdf ) / ( 1.0 - alpha_cdf );

  return cdf;
```

```
void truncated_normal_a_cdf_values ( int &n_data, double &mu, double &sigma,
  double &a, double &x, double &fx )
//  Purpose:
//    TRUNCATED_NORMAL_A_CDF_VALUES: values of the Lower Truncated Normal CDF.
{
# define N_MAX 11
  static double a_vec[N_MAX] = {50.0, 50.0, 50.0, 50.0, 50.0, 50.0,
     50.0, 50.0, 50.0, 50.0, 50.0 };
  static double fx_vec[N_MAX] = {0.3293202045481688, 0.3599223134505957, 0.3913175216041539,
    0.4233210140873113, 0.4557365629792204, 0.4883601253415709, 0.5209836877039214,
    0.5533992365958304, 0.5854027290789878, 0.6167979372325460, 0.6474000461349729 };
  static double mu_vec[N_MAX] = {100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,
     100.0, 100.0, 100.0, 100.0 };
  static double sigma_vec[N_MAX] = {25.0, 25.0, 25.0, 25.0, 25.0, 25.0, 25.0,
    25.0, 25.0, 25.0, 25.0  };
  static double x_vec[N_MAX] = {90.0, 92.0, 94.0, 96.0, 98.0, 100.0, 102.0,
    104.0, 106.0, 108.0, 110.0 };

  if ( n_data < 0 ) n_data = 0;
  n_data = n_data + 1;

  if ( N_MAX < n_data ) {
    n_data = 0;
    a = 0.0;
    mu = 0.0;
    sigma = 0.0;
    x = 0.0;
    fx = 0.0;
  } else {
    a = a_vec[n_data-1];
    mu = mu_vec[n_data-1];
    sigma = sigma_vec[n_data-1];
    x = x_vec[n_data-1];
    fx = fx_vec[n_data-1];
  }

  return;
# undef N MAX
}

double truncated_normal_a_cdf_inv ( double cdf, double mu, double s, double a )
//  Purpose:
//    TRUNCATED_NORMAL_A_CDF_INV inverts the lower truncated Normal CDF.
{
  double alpha;
  double alpha_cdf;
  double x;
  double xi;
  double xi_cdf;

  if ( cdf < 0.0 || 1.0 < cdf )  {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_A_CDF_INV - Fatal error!\n";
    cerr << "  CDF < 0 or 1 < CDF.\n";
    exit ( 1 );
  }
  alpha = ( a - mu ) / s;
  alpha_cdf = normal_01_cdf ( alpha );
  xi_cdf = ( 1.0 - alpha_cdf ) * cdf + alpha_cdf;
  xi = normal_01_cdf_inv ( xi_cdf );
  x = mu + s * xi;

  return x;
}
```

```
double truncated_normal_a_mean ( double mu, double s, double a )
//  Purpose:
//    TRUNCATED_NORMAL_A_MEAN returns the mean of the lower truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double alpha_pdf;
  double mean;

  alpha = ( a - mu ) / s;
  alpha_cdf = normal_01_cdf ( alpha );
  alpha_pdf = normal_01_pdf ( alpha );
  mean = mu + s * alpha_pdf / ( 1.0 - alpha_cdf );
  return mean;
}

double truncated_normal_a_moment ( int order, double mu, double s, double a )
//  Purpose:
//    TRUNCATED_NORMAL_A_MOMENT: moments of the lower truncated Normal PDF.
{
  double moment;
  moment = r8_mop ( order ) * truncated_normal_b_moment ( order, -mu, s, -a );

  return moment;
}

double truncated_normal_a_pdf ( double x, double mu, double s, double a )
//  Purpose:
//    TRUNCATED_NORMAL_A_PDF evaluates the lower truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double pdf;
  double xi;
  double xi_pdf;

  alpha = ( a - mu ) / s;
  xi = ( x - mu ) / s;
  alpha_cdf = normal_01_cdf ( alpha );
  xi_pdf = normal_01_pdf ( xi );
  pdf = xi_pdf / ( 1.0 - alpha_cdf ) / s;
  return pdf;
}

void truncated_normal_a_pdf_values ( int &n_data, double &mu, double &sigma,
  double &a, double &x, double &fx )
//  Purpose:
//    TRUNCATED_NORMAL_A_PDF_VALUES: values of the Lower Truncated Normal PDF.
{
# define N_MAX 11

  static double a_vec[N_MAX] = { 50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0,
     50.0, 50.0, 50.0, 50.0 };
  static double fx_vec[N_MAX] = {0.01507373507401876, 0.01551417047139894, 0.01586560931024694,
     0.01612150073158793, 0.01627701240029317, 0.01632918226724295, 0.01627701240029317,
     0.01612150073158793, 0.01586560931024694, 0.01551417047139894, 0.01507373507401876 };
  static double mu_vec[N_MAX] = {100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,
     100.0, 100.0, 100.0, 100.0 };
  static double sigma_vec[N_MAX] = {25.0, 25.0, 25.0, 25.0, 25.0, 25.0, 25.0,
    25.0, 25.0, 25.0, 25.0  };
  static double x_vec[N_MAX] = {90.0, 92.0, 94.0, 96.0, 98.0, 100.0, 102.0,
    104.0, 106.0, 108.0, 110.0 };

  if ( n_data < 0 ) n_data = 0;
  n_data = n_data + 1;
  if ( N_MAX < n_data ) {
    n_data = 0;
    a = 0.0;
    mu = 0.0;
    sigma = 0.0;
    x = 0.0;
    fx = 0.0;
  } else {
```

```
    a = a_vec[n_data-1];
    mu = mu_vec[n_data-1];
    sigma = sigma_vec[n_data-1];
    x = x_vec[n_data-1];
    fx = fx_vec[n_data-1];
  }

  return;
# undef N_MAX
}

double truncated_normal_a_sample ( double mu, double s, double a, int &seed )
//  Purpose:
//    TRUNCATED_NORMAL_A_SAMPLE samples the lower truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double u;
  double x;
  double xi;
  double xi_cdf;

  alpha = ( a - mu ) / s;
  alpha_cdf = normal_01_cdf ( alpha );
  u = r8_uniform_01 ( seed );
  xi_cdf = alpha_cdf + u * ( 1.0 - alpha_cdf );
  xi = normal_01_cdf_inv ( xi_cdf );
  x = mu + s * xi;

  return x;
}

double truncated_normal_a_variance ( double mu, double s, double a )
//  Purpose:
//    TRUNCATED_NORMAL_A_VARIANCE: variance of the lower truncated Normal PDF.
{
  double alpha;
  double alpha_cdf;
  double alpha_pdf;
  double variance;

  alpha = ( a - mu ) / s;
  alpha_pdf = normal_01_pdf ( alpha );
  alpha_cdf = normal_01_cdf ( alpha );
  variance = s * s * ( 1.0
    + alpha * alpha_pdf / ( 1.0 - alpha_cdf )
    - pow ( alpha_pdf / ( 1.0 - alpha_cdf ), 2 ) );
  return variance;
}

double truncated_normal_b_cdf ( double x, double mu, double s, double b )
//  Purpose:
//    TRUNCATED_NORMAL_B_CDF evaluates the upper truncated Normal CDF.
{
  double beta;
  double beta_cdf;
  double cdf;
  double xi;
  double xi_cdf;

  beta = ( b - mu ) / s;
  xi = ( x - mu ) / s;
  beta_cdf = normal_01_cdf ( beta );
  xi_cdf = normal_01_cdf ( xi );
  cdf = xi_cdf / beta_cdf;
  return cdf;
}
```

```
void truncated_normal_b_cdf_values ( int &n_data, double &mu, double &sigma,
  double &b, double &x, double &fx )
//  Purpose:
//    TRUNCATED_NORMAL_B_CDF_VALUES: values of the upper Truncated Normal CDF.
{
# define N_MAX 11
  static double b_vec[N_MAX] = {150.0, 150.0, 150.0, 150.0, 150.0,
     150.0, 150.0, 150.0, 150.0, 150.0, 150.0 };
  static double fx_vec[N_MAX] = {0.3525999538650271, 0.3832020627674540, 0.4145972709210122,
    0.4466007634041696, 0.4790163122960786, 0.5116398746584291, 0.5442634370207796,
    0.5766789859126887, 0.6086824783958461, 0.6400776865494043, 0.6706797954518312 };
  static double mu_vec[N_MAX] = {100.0, 100.0, 100.0, 100.0, 100.0,
     100.0, 100.0, 100.0, 100.0, 100.0, 100.0 };
  static double sigma_vec[N_MAX] = {25.0, 25.0, 25.0, 25.0, 25.0, 25.0, 25.0,
    25.0, 25.0, 25.0, 25.0  };
  static double x_vec[N_MAX] = {90.0, 92.0, 94.0, 96.0, 98.0, 100.0, 102.0,
    104.0, 106.0, 108.0, 110.0 };

  if ( n_data < 0 ) n_data = 0;
  n_data = n_data + 1;

  if ( N_MAX < n_data ) {
    n_data = 0;
    b = 0.0;
    mu = 0.0;
    sigma = 0.0;
    x = 0.0;
    fx = 0.0;
  } else {
    b = b_vec[n_data-1];
    mu = mu_vec[n_data-1];
    sigma = sigma_vec[n_data-1];
    x = x_vec[n_data-1];
    fx = fx_vec[n_data-1];
  }

  return;
# undef N_MAX
}
double truncated_normal_b_cdf_inv ( double cdf, double mu, double s, double b )
//  Purpose:
//    TRUNCATED_NORMAL_B_CDF_INV inverts the upper truncated Normal CDF.
{
  double beta;
  double beta_cdf;
  double x;
  double xi;
  double xi_cdf;

  if ( cdf < 0.0 || 1.0 < cdf ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_B_CDF_INV - Fatal error!\n";
    cerr << "  CDF < 0 or 1 < CDF.\n";
    exit ( 1 );
  }
  beta = ( b - mu ) / s;
  beta_cdf = normal_01_cdf ( beta );
  xi_cdf = beta_cdf * cdf;
  xi = normal_01_cdf_inv ( xi_cdf );
  x = mu + s * xi;

  return x;
}

double truncated_normal_b_mean ( double mu, double s, double b )
//  Purpose:
//    TRUNCATED_NORMAL_B_MEAN returns the mean of the upper truncated Normal PDF.
{
  double beta;
  double beta_cdf;
  double beta_pdf;
  double mean;
  beta = ( b - mu ) / s;
  beta_cdf = normal_01_cdf ( beta );
  beta_pdf = normal_01_pdf ( beta );
  mean = mu - s * beta_pdf / beta_cdf;
  return mean;
}
```

```
double truncated_normal_b_moment ( int order, double mu, double s, double b )
//  Purpose:
//    TRUNCATED_NORMAL_B_MOMENT: moments of the upper truncated Normal PDF.
{
  double f, h, h_cdf, h_pdf, ir, irm1, irm2, moment;
  int r;

  if ( order < 0 ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_B_MOMENT - Fatal error!\n";
    cerr << "  ORDER < 0.\n";
    exit ( 1 );
  }

  h = ( b - mu ) / s;
  h_pdf = normal_01_pdf ( h );
  h_cdf = normal_01_cdf ( h );

  if ( h_cdf == 0.0 ) {
    cerr << "\n";
    cerr << "TRUNCATED_NORMAL_B_MOMENT - Fatal error!\n";
    cerr << "  CDF((B-MU)/S) = 0.\n";
    exit ( 1 );
  }

  f = h_pdf / h_cdf;
  moment = 0.0;
  irm2 = 0.0;
  irm1 = 0.0;

  for ( r = 0; r <= order; r++ ) {
    if ( r == 0 ) {
      ir = 1.0;
    } else if ( r == 1 ) {
      ir = - f;
    } else {
      ir = - pow ( h, r - 1 ) * f + ( double ) ( r - 1 ) * irm2;
    }

    moment = moment + r8_choose ( order, r ) * pow ( mu, order - r )
      * pow ( s, r ) * ir;
    irm2 = irm1;
    irm1 = ir;
  }
  return moment;
}

double truncated_normal_b_pdf ( double x, double mu, double s, double b )
//  Purpose:
//    TRUNCATED_NORMAL_B_PDF evaluates the upper truncated Normal PDF.
{
  double beta, beta_cdf, pdf, xi, xi_pdf;

  beta = ( b - mu ) / s;
  xi = ( x - mu ) / s;
  beta_cdf = normal_01_cdf ( beta );
  xi_pdf = normal_01_pdf ( xi );
  pdf = xi_pdf / beta_cdf / s;
  return pdf;
}
```

```
void truncated_normal_b_pdf_values ( int &n_data, double &mu, double &sigma,
  double &b, double &x, double &fx )
//  Purpose:
//    TRUNCATED_NORMAL_B_PDF_VALUES: values of the Upper Truncated Normal PDF.
{
# define N_MAX 11
  static double b_vec[N_MAX] = { 150.0, 150.0, 150.0, 150.0, 150.0, 150.0, 150.0,
     150.0, 150.0, 150.0, 150.0 };
  static double fx_vec[N_MAX] = {0.01507373507401876, 0.01551417047139894, 0.01586560931024694,
    0.01612150073158793, 0.01627701240029317, 0.01632918226724295, 0.01627701240029317,
    0.01612150073158793, 0.01586560931024694, 0.01551417047139894, 0.01507373507401876 };
  static double mu_vec[N_MAX] = {100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 100.0,
     100.0, 100.0, 100.0, 100.0 };
  static double sigma_vec[N_MAX] = {25.0, 25.0, 25.0, 25.0, 25.0, 25.0, 25.0,
    25.0, 25.0, 25.0, 25.0  };
  static double x_vec[N_MAX] = {90.0, 92.0, 94.0, 96.0, 98.0, 100.0, 102.0,
    104.0, 106.0, 108.0, 110.0 };

  if ( n_data < 0) n_data = 0;
  n_data = n_data + 1;

  if ( N_MAX < n_data ) {
    n_data = 0;
    b = 0.0;
    mu = 0.0;
    sigma = 0.0;
    x = 0.0;
    fx = 0.0;
  } else {
    b = b_vec[n_data-1];
    mu = mu_vec[n_data-1];
    sigma = sigma_vec[n_data-1];
    x = x_vec[n_data-1];
    fx = fx_vec[n_data-1];
  }

  return;
# undef N_MAX
}
double truncated_normal_b_sample ( double mu, double s, double b, int &seed )
//  Purpose:
//    TRUNCATED_NORMAL_B_SAMPLE samples the upper truncated Normal PDF.
{
  double beta, beta_cdf, u, x, xi, xi_cdf;

  beta = ( b - mu ) / s;
  beta_cdf = normal_01_cdf ( beta );
  u = r8_uniform_01 ( seed );
  xi_cdf = u * beta_cdf;
  xi = normal_01_cdf_inv ( xi_cdf );
  x = mu + s * xi;
  return x;
}

double truncated_normal_b_variance ( double mu, double s, double b )
//  Purpose:
//    TRUNCATED_NORMAL_B_VARIANCE: variance of the upper truncated Normal PDF.
{
  double beta, beta_cdf, beta_pdf, variance;
  beta = ( b - mu ) / s;
  beta_pdf = normal_01_pdf ( beta );
  beta_cdf = normal_01_cdf ( beta );
  variance = s * s * ( 1.0
    - beta * beta_pdf / beta_cdf
    - pow ( beta_pdf / beta_cdf, 2 ) );
  return variance;
}
```

Figura C.3: truncated$_n ormal.cpp$

```cpp
#include "r_node.h"
#include <iostream>
#include "truncated_normal.hpp"

int RNode::current_id;

RNode::RNode() : id(current_id++) {
    evidence = 9; // Clear observation - change to -1
}

RNode::RNode(int id) {
    this->id = id;
    _nStates = 0;
    _mu = DEFAULT_MU;
    _var = DEFAULT_VAR;
    evidence = 9; // Clear observation
    _samples.reserve(100000);
    _samples.resize(100000);
}

// Remover esses construtores sobrecarregados depois
RNode::RNode(string state1, double inVar, double inMu) {
    Init1StateNode(state1, inVar, inMu);
}

RNode::RNode(string state1, string state2, double inVar, double inMu) {
    Init2StateNode(state1, state2, inVar, inMu);
}

RNode::RNode(string state1, string state2, string state3, double inVar, double inMu) {
    Init3StateNode(state1, state2, state3, inVar, inMu);
}

RNode::RNode(string state1, string state2, string state3, string state4, double inVar, double inMu) {
    Init4StateNode(state1, state2, state3, state4, inVar, inMu);
}

RNode::RNode(string state1, string state2, string state3, string state4, string state5, double inVar, double inMu) {
    Init5StateNode(state1, state2, state3, state4, state5, inVar, inMu);
}
void RNode::Init1StateNode(string state1, double inVar, double inMu) {
    _nStates = 1;
    _mu = inMu;
    _var = inVar;

    _samples.reserve(100000);
    _samples.resize(100000);

    _stateTitles.reserve(1);
    _stateTitles.resize(1);

    _stateValues.reserve(1);
    _stateValues.resize(1);

    _stateTitles.at(0) = state1.c_str();

    _stateValues.at(0) = 1;

    _stateIntervals.clear();
    // Mapping of state intervals
    for (int i = 1; i <= _nStates; i++)
        _stateIntervals.push_back((double) i / _nStates);
}

void RNode::Init2StateNode(string state1, string state2, double inVar, double inMu) {
    _nStates = 2;
    _mu = inMu;
    _var = inVar;

    _samples.reserve(100000);
    _samples.resize(100000);

    _stateTitles.reserve(2);
    _stateTitles.resize(2);

    _stateValues.reserve(2);
    _stateValues.resize(2);

    _stateTitles.at(0) = state1.c_str();
    _stateTitles.at(1) = state2.c_str();
```

```
    _stateValues.at(0) = .5;
    _stateValues.at(1) = .5;

    _stateIntervals.clear();
    // Mapping of state intervals
    for (int i = 1; i <= _nStates; i++)
        _stateIntervals.push_back((double) i / _nStates);
}

void RNode::Init3StateNode(string state1, string state2, string state3, double inVar, double inMu) {
    _nStates = 3;
    _mu = inMu;
    _var = inVar;

    _samples.reserve(100000);
    _samples.resize(100000);

    _stateTitles.reserve(3);
    _stateTitles.resize(3);

    _stateValues.reserve(3);
    _stateValues.resize(3);

    _stateTitles.at(0) = state1.c_str();
    _stateTitles.at(1) = state2.c_str();
    _stateTitles.at(2) = state3.c_str();

    _stateValues.at(0) = .33333;
    _stateValues.at(1) = .33333;
    _stateValues.at(2) = .33333;

    _stateIntervals.clear();
    // Mapping of state intervals
    for (int i = 1; i <= _nStates; i++)
        _stateIntervals.push_back((double) i / _nStates);
}
void RNode::Init4StateNode(string state1, string state2, string state3, string state4, double inVar, double inMu) {
    _nStates = 4;
    _mu = inMu;
    _var = inVar;

    _samples.reserve(100000);
    _samples.resize(100000);

    _stateTitles.reserve(4);
    _stateTitles.resize(4);

    _stateValues.reserve(4);
    _stateValues.resize(4);

    _stateTitles.at(0) = state1.c_str();
    _stateTitles.at(1) = state2.c_str();
    _stateTitles.at(2) = state3.c_str();
    _stateTitles.at(3) = state4.c_str();

    _stateValues.at(0) = .25;
    _stateValues.at(1) = .25;
    _stateValues.at(2) = .25;
    _stateValues.at(3) = .25;

    _stateIntervals.clear();
    // Mapping of state intervals
    for (int i = 1; i <= _nStates; i++)
        _stateIntervals.push_back((double) i / _nStates);
}
```

```
void RNode::Init5StateNode(string state1, string state2, string state3, string state4, string state5, double inVar, double inMu) {
    _nStates = 5;
    _mu = inMu;
    _var = inVar;

    _samples.reserve(100000);
    _samples.resize(100000);

    _stateTitles.reserve(5);
    _stateTitles.resize(5);

    _stateValues.reserve(5);
    _stateValues.resize(5);

    _stateTitles.at(0) = state1.c_str();
    _stateTitles.at(1) = state2.c_str();
    _stateTitles.at(2) = state3.c_str();
    _stateTitles.at(3) = state4.c_str();
    _stateTitles.at(4) = state5.c_str();

    _stateValues.at(0) = .20000;
    _stateValues.at(1) = .20000;
    _stateValues.at(2) = .20000;
    _stateValues.at(3) = .20000;
    _stateValues.at(4) = .20000;

    _stateIntervals.clear();        // Whatch out for this, sometimes is better swap
    // Mapping of state intervals
    for (int i = 1; i <= _nStates; i++)
        _stateIntervals.push_back((double) i / _nStates);
}

RNode::~RNode(){}

void RNode::addParent(shared_ptr<RNode> parentNode, double weight) {
    setParent(parentNode);
    setParentWeight(weight);
    parentNode->setChild(shared_from_this());
}
void RNode::addParentMixMinMax(shared_ptr<RNode> parentNode) {
    setParent(parentNode);
    parentNode->setChild(shared_from_this());
}

void RNode::setChild(shared_ptr<RNode> childNode) {
    _childNodes.push_back(childNode);
}

void RNode::setParent(shared_ptr<RNode> parentNode) {
    _parents.push_back(parentNode);
}

void RNode::setParentWeight(double weight) {
    _parentsWeight.push_back(weight);
}

// Setters
void RNode::setMean(double inMu) {
    _mu = inMu;
}

void RNode::setVariance(double inVar) {
    _var = inVar;
}

void RNode::setTPN(vector<double> tpn) {
    _tpn = tpn;
}

void RNode::setSamples(vector<double> samples) {
    _samples = samples;
}

void RNode::setName(string name) {
    _name = name;
}
```

```
// Getters
bool RNode::isChild() {
    return _parents.size() > 0;
}

vector<double>* RNode::getTpnPtr() {
    return &_tpn;
}

vector<string>* RNode::getStateTitlesPtr() {
    return &_stateTitles;
}

vector<shared_ptr<RNode>>* RNode::getParentsPtr() {
    return &_parents;
}

vector<double>* RNode::getStateValuesPtr() {
    return &_stateValues;
}

vector<double>* RNode::getStateIntervalsPtr() {
    return &_stateIntervals;
}

vector<double>* RNode::getSamplesPtr() {
    return &_samples;
}

int RNode::getStatesQuantity() {
    return _nStates;
}

double RNode::getMean() {
    return _mu;
}

double RNode::getVariance() {
    return _var;
}
int RNode::getSize() {
    return _parents.size();
}

vector<double>* RNode::getParentsWeightPtr() {
    return &_parentsWeight;
}

string RNode::getName() {
    return _name;
}

vector<shared_ptr<RNode>>* RNode::getChildNodesPtr() {
    return &_childNodes;
}

vector<shared_ptr<RNode>> RNode::getParents() {
    return _parents;
}

void RNode::generateEvidence(double inMu) {
    int seed = 123456789;
    double aux;

    for (int i = 0; i < SAMPLESIZE; i++) {
        aux = truncated_normal_ab_sample(inMu, .0001, LOWER_BOUND, UPPER_BOUND, seed);
        _samples.at(i) = aux;
    }
}

void RNode::generateSamples(double mean) {
    int seed = 123456789;
    double aux;

    for (int i = 0; i < _nStates; ++i)
        _stateValues.at(i) = 0;
```

```
    for (int i = 0; i < SAMPLESIZE; i++) {
        aux = truncated_normal_ab_sample(mean, .0001, LOWER_BOUND, UPPER_BOUND, seed);
        _samples.at(i) = aux;
        if (_samples.at(i) < _stateIntervals.at(0)) {
            _stateValues.at(0) += 1;
        } else if (_samples.at(i) < _stateIntervals.at(1)) {
            _stateValues.at(1) += 1;
        }
        else if (_samples.at(i) < _stateIntervals.at(2)) {
            _stateValues.at(2) += 1;
        } else if (_nStates > 3) {
            if (_samples.at(i) < _stateIntervals.at(3)) {
                _stateValues.at(3) += 1;
            } else {
                _stateValues.at(4) += 1;
            }
        }
    }

    _stateValues.at(0) = (double) _stateValues.at(0) / SAMPLESIZE ;

    if(_nStates > 1)
        _stateValues.at(1) = (double) _stateValues.at(1) / SAMPLESIZE;

    if(_nStates > 2)
        _stateValues.at(2) = (double) _stateValues.at(2) / SAMPLESIZE;

    if (_nStates > 3)
        _stateValues.at(3) = (double) _stateValues.at(3) / SAMPLESIZE;

    if (_nStates > 4)
        _stateValues.at(4) = (double) _stateValues.at(4) / SAMPLESIZE;
}

// Util
void RNode::printValues() {
    cout << _name << endl;
    cout << "---------------------" << endl;
    for (int i = 0; i < _nStates; i++)
        cout << _stateTitles.at(i) << ": " << _stateValues.at(i) << endl;
    cout << "---------------------\n" << endl;
}

void RNode::createMean(int evidence) {
    if(getStatesQuantity() == 1) {
        _mu = (double) getStateIntervalsPtr()->at(evidence) - .500000;
    } else if(getStatesQuantity() == 2) {
        _mu = (double) getStateIntervalsPtr()->at(evidence) - .250000;
    } else if(getStatesQuantity() == 3) {
        _mu = (double) getStateIntervalsPtr()->at(evidence) - .166667;
    } else if(getStatesQuantity() == 4) {
        _mu = (double) getStateIntervalsPtr()->at(evidence) - .125000;
    } else if(getStatesQuantity() == 5) {
        _mu = (double) getStateIntervalsPtr()->at(evidence) - .1;
    }
}

// Core
void RNode::ResetStateValues() {
    switch(_nStates)    {
    case (1):
        _stateValues.at(0) = 1;
        break;
    case (2):
        _stateValues.at(0) = .5;
        _stateValues.at(1) = .5;
        break;
    case (3):
        _stateValues.at(0) = .33;
        _stateValues.at(1) = .33;
        _stateValues.at(2) = .33;
        break;
    case (4):
        _stateValues.at(0) = .25;
        _stateValues.at(1) = .25;
        _stateValues.at(2) = .25;
        _stateValues.at(3) = .25;
        break;
```

```
    case (5):
        _stateValues.at(0) = .20;
        _stateValues.at(1) = .20;
        _stateValues.at(2) = .20;
        _stateValues.at(3) = .20;
        _stateValues.at(4) = .20;
        break;
    }
}

/* Raissa */
void RNode::wmean() {
    double div = 0;
    double sum = 0;
    double mean = 0;
    double aux = 0;
    int seed = 123456789;

    // Descobre por quem devo dividir
    for (int i = 0; i < _parents.size(); i++){
        div += _parentsWeight.at(i);
    }

    int counter = 0;

    // WMean
    /* Percorre o vetor de amostras utilizando passo 10.
       SAMPLESIZE : Tamanho das amostras definidas por Mirko 100k.
       ESQUIDISTANT_PASS : Com valor 10 implica em 10k iterações capturando amostras equidistantes e misturando-as.
    */
    for (int i = 0; i < SAMPLESIZE; i+=EQUIDISTANT_PASS) {
        // Pega uma amostra de cada pai multiplicando pelo peso e mistura.
        // Em um nó com 3 pais teremos 3 iterações capturando 3 amostras equidistantes de cada pai, somando e guardando em "sum".
        for (int j = 0; j < _parents.size(); j++) {
            // Se o nó tiver 3 pais, vai rodar 3x
            // _parents.at(j) pega o pai atual
            // _parents.at(j)->getSamplesPtr()->at(i) pega uma das amostras do nó pai em questão
            // _parents.at(j)->getSamplesPtr()->at(i) * _parentsWeight.at(j); multiplica pelo peso
            sum += _parents.at(j)->getSamplesPtr()->at(i) * _parentsWeight.at(j);
        }
        // Pego a média da mistura das amostras.
        mean = sum / div;

        // Prepararo sum para a próxima iteração.
        sum = 0;

        // Em aux armazeno o valor truncado da média levando em consideração a variância.
        // O método pra fazer isso está na biblioteca trunc_norm.
        aux = truncated_normal_ab_sample(mean, _var, 0,
        1, seed);

        // _samples guarda amostras do nó filho.
        // Nesse caso counter é usado por que o loop principal em que estamos dentro ta pulando de 10 em 10.
        // As amostras no nó filho são de 10k - Só preciso de amostras grandes para as evidências "base".
        _samples.at(counter) = aux;

        // Guarda quantas vezes as amostras ficaram no intervalo [0, 1/3], [1/3, 2/3] etc.
        if (_samples.at(counter) < _stateIntervals.at(0)) {
            _stateValues.at(0) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(1)) {
            _stateValues.at(1) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(2)) {
            _stateValues.at(2) += 1;
        } else if (_nStates > 3) {
            if (_samples.at(counter) < _stateIntervals.at(3)) {
                _stateValues.at(3) += 1;
            } else {
                _stateValues.at(4) += 1;
            }
        }
        counter++;
    }

    // Normaliza esse valor dividindo por 10k
    _stateValues.at(0) = (double) _stateValues.at(0) / EQUIDISTANT_SAMPLESIZE;

    if(_nStates > 1)
        _stateValues.at(1) = (double) _stateValues.at(1) / EQUIDISTANT_SAMPLESIZE;
```

```
    if(_nStates > 2)
        _stateValues.at(2) = (double) _stateValues.at(2) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 3)
        _stateValues.at(3) = (double) _stateValues.at(3) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 4)
        _stateValues.at(4) = (double) _stateValues.at(4) / EQUIDISTANT_SAMPLESIZE;

    // Preenche a tabela de probabilidades
    _tpn.push_back(_stateValues.at(0));

    if(_nStates > 1)
        _tpn.push_back(_stateValues.at(1));
    if(_nStates > 2)
        _tpn.push_back(_stateValues.at(2));
    if (_nStates > 3)
        _tpn.push_back(_stateValues.at(3));
    if (_nStates > 4)
        _tpn.push_back(_stateValues.at(4));
}

/* Raissa */
void RNode::wmin() {
    double highestValue;
    double brierScore;
    double div = 0;
    double sum = 0;
    double aux = 0;
    double min = 123456789;
    double a = 0;
    double currentMin = 0;
    int seed = 123456789;

    for (int i = 0; i < SAMPLESIZE; i+=EQUIDISTANT_PASS) {
        // Pega uma amostra de cada pai multiplicando pelo peso e mistura.
        // Em um nó com 3 pais teremos 3 iterações capturando 3 amostras equidistantes de cada pai, somando e guardando em "sum".
        for (int j = 0; j < _parents.size(); j++) {
            // Se o nó tiver 3 pais, vai rodar 3x
            // _parents.at(j) pega o pai atual
            // _parents.at(j)->getSamplesPtr()->at(i) pega uma das amostras do nó pai em questão
            // _parents.at(j)->getSamplesPtr()->at(i) * _parentsWeight.at(j); multiplica pelo peso
            div = _parentsWeight.at(j) + _parents.size() - 1;
            a = _parents.at(j)->getSamplesPtr()->at(i) * _parentsWeight.at(j);
            for (int k = 0; k < _parents.size(); k++) {
                if (k != j) {
                    sum += _parents.at(k)->getSamplesPtr()->at(i);
                }
            }

            currentMin = (a + sum) / div;
            if (currentMin < min) {
                min = currentMin;
            }

        sum = 0;
        }

        // Prepararo sum para a próxima iteração.

        // Em aux armazeno o valor truncado da média levando em consideração a variância.
        // O método pra fazer isso está na biblioteca trunc_norm.
        aux = truncated_normal_ab_sample(min, _var, 0,
        1, seed);

        int counter = 0;

        // _samples guarda amostras do nó filho.
        // Nesse caso counter é usado por que o loop principal em que estamos dentro ta pulando de 10 em 10.
        // As amostras no nó filho são de 10k - Só preciso de amostras grandes para as evidências "base".
        _samples.at(counter) = aux;

        // Guarda quantas vezes as amostras ficaram no intervalo [0, 1/3], [1/3, 2/3] etc.
        if (_samples.at(counter) < _stateIntervals.at(0)) {
            _stateValues.at(0) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(1)) {
            _stateValues.at(1) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(2)) {
            _stateValues.at(2) += 1;
```

```
        } else if (_nStates > 3) {
            if (_samples.at(counter) < _stateIntervals.at(3)) {
                _stateValues.at(3) += 1;
            } else {
                _stateValues.at(4) += 1;
            }
        }
        counter++;
    }

    // Normaliza esse valor dividindo por 10k
    _stateValues.at(0) = (double) _stateValues.at(0) / EQUIDISTANT_SAMPLESIZE;

    if(_nStates > 1)
        _stateValues.at(1) = (double) _stateValues.at(1) / EQUIDISTANT_SAMPLESIZE;
    if(_nStates > 2)
        _stateValues.at(2) = (double) _stateValues.at(2) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 3)
        _stateValues.at(3) = (double) _stateValues.at(3) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 4)
        _stateValues.at(4) = (double) _stateValues.at(4) / EQUIDISTANT_SAMPLESIZE;

    // Preenche a tabela de probabilidades
    _tpn.push_back(_stateValues.at(0));

    if(_nStates > 1)
        _tpn.push_back(_stateValues.at(1));
    if(_nStates > 2)
        _tpn.push_back(_stateValues.at(2));
    if (_nStates > 3)
        _tpn.push_back(_stateValues.at(3));
    if (_nStates > 4)
        _tpn.push_back(_stateValues.at(4));
}

/* Raissa */
void RNode::wmax()
{
    double highestValue;
    double brierScore;
    double div = 0;
    double sum = 0;
    double aux = 0;
    double max = 0;
    double a = 0;
    double currentMax = 0;
    int seed = 123456789;

    for (int i = 0; i < SAMPLESIZE; i+=EQUIDISTANT_PASS)
    {

        // Pega uma amostra de cada pai multiplicando pelo peso e mistura.
        // Em um nó com 3 pais teremos 3 iterações capturando 3 amostras equidistantes de cada pai, somando e guardando em "sum".
        for (int j = 0; j < _parents.size(); j++)
        {
            // Se o nó tiver 3 pais, vai rodar 3x
            // _parents.at(j) pega o pai atual
            // _parents.at(j)->getSamplesPtr()->at(i) pega uma das amostras do nó pai em questão
            // _parents.at(j)->getSamplesPtr()->at(i) * _parentsWeight.at(j); multiplica pelo peso
            div = _parentsWeight.at(j) + _parents.size() - 1;
            a = _parents.at(j)->getSamplesPtr()->at(i) * _parentsWeight.at(j);
            for (int k = 0; k < _parents.size(); k++) {
                if (k != j) {
                    sum += _parents.at(k)->getSamplesPtr()->at(i);
                }
            }
```

```
            currentMax = (a + sum) / div;
            if (currentMax > max)  max = currentMax;
        sum = 0;
        }

        // Prepararo sum para a próxima iteração.
        // Em aux armazeno o valor truncado da média levando em consideração a variância.
        // O método pra fazer isso está na biblioteca trunc_norm.
        aux = truncated_normal_ab_sample(max, _var, 0,
        1, seed);

        int counter = 0;
        // _samples guarda amostras do nó filho.
        // Nesse caso counter é usado por que o loop principal em que estamos dentro ta pulando de 10 em 10.
        // As amostras no nó filho são de 10k - Só preciso de amostras grandes para as evidências "base".
        _samples.at(counter) = aux;

        // Guarda quantas vezes as amostras ficaram no intervalo [0, 1/3], [1/3, 2/3] etc.
        if (_samples.at(counter) < _stateIntervals.at(0)) {
            _stateValues.at(0) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(1)) {
            _stateValues.at(1) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(2)) {
            _stateValues.at(2) += 1;
        } else if (_nStates > 3) {
            if (_samples.at(counter) < _stateIntervals.at(3)) {
                _stateValues.at(3) += 1;
            } else {
                _stateValues.at(4) += 1;
            }
        }
        counter++;
    }
    // Normaliza esse valor dividindo por 10k
    _stateValues.at(0) = (double) _stateValues.at(0) / EQUIDISTANT_SAMPLESIZE;
    if(_nStates > 1)
        _stateValues.at(1) = (double) _stateValues.at(1) / EQUIDISTANT_SAMPLESIZE;
    if(_nStates > 2)
        _stateValues.at(2) = (double) _stateValues.at(2) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 3)
        _stateValues.at(3) = (double) _stateValues.at(3) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 4)
        _stateValues.at(4) = (double) _stateValues.at(4) / EQUIDISTANT_SAMPLESIZE;

    // Preenche a tabela de probabilidades
    _tpn.push_back(_stateValues.at(0));
    if (_nStates > 1) _tpn.push_back(_stateValues.at(1));
    if (_nStates > 2) _tpn.push_back(_stateValues.at(2));
    if (_nStates > 3) _tpn.push_back(_stateValues.at(3));
    if (_nStates > 4) _tpn.push_back(_stateValues.at(4));
}

 /* Raissa */
void RNode::mixMinMax(double wmin, double wmax)
{
    double sum = 0;
    double aux = 0;
    double max = 0;
    double min = 123456789;
    double a = 0;
    double w = 0;
    double wminFunction = 0;
    double div = 0;
    double currentMin = 0;
    double currentMax = 0;
    double mixMinMax = 0;
    int seed = 123456789;

    for (int i = 0; i < SAMPLESIZE; i+=EQUIDISTANT_PASS) {

        // Pega uma amostra de cada pai multiplicando pelo peso e mistura.
        // Em um nó com 3 pais teremos 3 iterações capturando 3 amostras equidistantes de cada pai, somando e guardando em "sum".
        for (int j = 0; j < _parents.size(); j++) {

            div = _parentsWeight.at(j) + _parents.size() - 1;
```

```
            a = _parents.at(j)->getSamplesPtr()->at(i) * _parentsWeight.at(j);
            for (int k = 0; k < _parents.size(); k++) {
                if (k != j) {
                    sum += _parents.at(k)->getSamplesPtr()->at(i);
                }
            }

            currentMin = (a + sum) / div;
            if (currentMin < min) {
                min = currentMin;
            }

            currentMax = (a + sum) / div;
            if (currentMax > max) {
                max = currentMax;
            }
        sum = 0;

        }
        //div += wmin + wmax;
        mixMinMax = (wmin * min + wmax * max) / wmin+wmax;

        // Prepararo sum para a próxima iteração.
        // Em aux armazeno o valor truncado da média levando em consideração a variância.
        // O método pra fazer isso está na biblioteca trunc_norm.
        aux = truncated_normal_ab_sample(mixMinMax, _var, 0,
        1, seed);

        int counter = 0;
        // _samples guarda amostras do nó filho.
        // Nesse caso counter é usado por que o loop principal em que estamos dentro ta pulando de 10 em 10.
        // As amostras no nó filho são de 10k - Só preciso de amostras grandes para as evidências "base".
        _samples.at(counter) = aux;

        // Guarda quantas vezes as amostras ficaram no intervalo [0, 1/3], [1/3, 2/3] etc.
        if (_samples.at(counter) < _stateIntervals.at(0)) {
            _stateValues.at(0) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(1)) {
            _stateValues.at(1) += 1;
        } else if (_samples.at(counter) < _stateIntervals.at(2)) {
            _stateValues.at(2) += 1;
        } else if (_nStates > 3) {
            if (_samples.at(counter) < _stateIntervals.at(3)) {
                _stateValues.at(3) += 1;
            } else {
                _stateValues.at(4) += 1;
            }
        }
        counter++;
    }

    // Normaliza esse valor dividindo por 10k
    _stateValues.at(0) = (double) _stateValues.at(0) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 1) _stateValues.at(1) = (double) _stateValues.at(1) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 2) _stateValues.at(2) = (double) _stateValues.at(2) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 3) _stateValues.at(3) = (double) _stateValues.at(3) / EQUIDISTANT_SAMPLESIZE;
    if (_nStates > 4) _stateValues.at(4) = (double) _stateValues.at(4) / EQUIDISTANT_SAMPLESIZE;

    // Preenche a tabela de probabilidades
    _tpn.push_back(_stateValues.at(0));
    if (_nStates > 1) _tpn.push_back(_stateValues.at(1));
    if (_nStates > 2) _tpn.push_back(_stateValues.at(2));
    if (_nStates > 3) _tpn.push_back(_stateValues.at(3));
    if (_nStates > 4) _tpn.push_back(_stateValues.at(4));
}

void RNode::removeParent(shared_ptr<RNode> parent) {
    for(int i = 0; i < _parents.size(); i++) {
        if(_parents.at(i) == parent) {
            _parents.at(i)->removeChild(shared_from_this());
            _parents.erase(_parents.begin()+i);
            _parentsWeight.erase(_parentsWeight.begin()+i);
        }
    }
}
```

```
void RNode::removeChild(shared_ptr<RNode> child) {
    for(int i = 0; i < _childNodes.size(); i++) {
        if(_childNodes.at(i) == child) {
            _childNodes.erase(_childNodes.begin()+i);
        }
    }
}

void generateBestResult(float variance) {}

void RNode::updateTPN() {
    //vertigo();
}

void RNode::setEvidence(int stateEvidence) {
    evidence = stateEvidence;
}

int RNode::getEvidence() {
    return evidence;
}

int RNode::getId() {
    return id;
}

void RNode::addParentId(int id) {
    _parentsId.push_back(id);
}

void RNode::addChildId(int id) {
    _childrenId.push_back(id);
}

int RNode::getParentId(int parentIndex) {
    return _parentsId.at(parentIndex);
}

int RNode::getChildId(int childIndex) {
    return _childrenId.at(childIndex);
}

vector<int> RNode::getParentsIdVector() {
    return _parentsId;
}

void RNode::init() {
    _nStates = _stateTitles.size();

    _samples.reserve(100000);
    _samples.resize(100000);

    _stateValues.reserve(_nStates);
    _stateValues.resize(_nStates);

    switch (_nStates) {
    case 1:
        _stateValues.at(0) = 1;
        break;
    case 2:
        _stateValues.at(0) = .5;
        _stateValues.at(1) = .5;
        break;
    case 3:
        _stateValues.at(0) = .33333;
        _stateValues.at(1) = .33333;
        _stateValues.at(2) = .33333;
        break;
    case 4:
        _stateValues.at(0) = .25;
        _stateValues.at(1) = .25;
        _stateValues.at(2) = .25;
        _stateValues.at(3) = .25;
        break;
```

```
    default:
        _stateValues.at(0) = .20000;
        _stateValues.at(1) = .20000;
        _stateValues.at(2) = .20000;
        _stateValues.at(3) = .20000;
        _stateValues.at(4) = .20000;
    }

    _stateIntervals.clear();
    // Mapping of state intervals
    for (int i = 1; i <= _nStates; i++)
        _stateIntervals.push_back((double) i / _nStates);
}

shared_ptr<RNode>RNode::withState(string name) {
    if(_stateTitles.size() <= 5)
        _stateTitles.push_back(name);
    return shared_from_this();
}

shared_ptr<RNode> RNode::withMean(double inMu) {
    _mu = inMu;
    return shared_from_this();
}

shared_ptr<RNode> RNode::withVariance(double inVar) {
    _var = inVar;
    return shared_from_this();
}

shared_ptr<RNode> RNode::withName(string nodeName) {
    _name = nodeName;
    return shared_from_this();
}
```

Figura C.4: rnode.cpp

Apêndice D

Código Método Genético

```
float variance;
char state[2];

int main() {

    string fileName;

    // get file name
    fileName = "teste.txt";
    srand (time(0));
    // start genetic algorithm
    GeneticAlgorithm* ga = new GeneticAlgorithm(fileName);

    // executa genetic algorithm
    ga->evolve();

    getch();
    return 0;
}
```

Figura D.1: main.cpp

```
#include "geneticAlgorithm.h"
#include<memory>

int popsize = 50;
int maxgens = 10;
int nvars = 3;
double xover = 0.8;
double pmutation = 0.3;

std::vector<shared_ptr<GenomeRankedNode>> population;
std::vector<shared_ptr<GenomeRankedNode>> newpopulation;
std::vector<double> fitness;
std::vector<double> rfitness;
std::vector<double> cfitness;
string fileName;
BrierScore* brierScore;

GeneticAlgorithm::GeneticAlgorithm(string inFileName) {
    fileName = inFileName;
    brierScore = new BrierScore(fileName);
}

GeneticAlgorithm::~GeneticAlgorithm(void) {}

void GeneticAlgorithm::evolve()
//  Purpose:
//    MAIN supervises the genetic algorithm.
//
//  Discussion:
//
//    Each generation involves selecting the best
//    members, performing crossover & mutation and then
//    evaluating the resulting population, until the terminating
//    condition is satisfied
//
//    This is a simple genetic algorithm implementation where the
//    evaluation function takes positive values only and the
//    fitness of an individual is the same as the value of the
//    objective function.
{
  int generation;
  int i;
  int seed;
  newpopulation.resize(popsize);

  timestamp ( );
  std::cout << "\n";
  std::cout << "Running the genetic algorithm.\n";
  seed = 123456789;

  initialize();
  evaluate();
  keep_the_best();

  for ( generation = 0; generation < maxgens; generation++ ) {
    selector ( seed );
    crossover ( seed );
    mutate ( seed );
    report ( generation );
    evaluate ( );
    elitist ( );
    if (fitness[popsize - 1] < 0.1) {
        break;
    }
  }

  std::cout << "\n";
  std::cout << "  Best member after " << generation << " generations:\n";
  std::cout << "\n";
```

```
  std::cout << "\n";
  std::cout << "  Best fitness = " << fitness[popsize-1] << "\n";
  std::cout << "\n";
  std::cout << "  Normal end of execution.\n";
  std::cout << "\n";
  timestamp ( );
}

void GeneticAlgorithm::crossover ( int &seed )
//  Purpose:
//    CROSSOVER selects two parents for the single point crossover.
{
  const double a = 0.0;
  const double b = 1.0;
  int mem;
  int one;
  int first = 0;
  double x;

  for ( mem = 0; mem < popsize; ++mem ) {
    x = r8_uniform_ab ( a, b, seed );

    if ( x < xover ) {
      ++first;

      if ( first % 2 == 0 ) {
        Xover ( one, mem, seed );
      } else {
        one = mem;
      }
    }
  }
  return;
}

void GeneticAlgorithm::elitist ( )
//  Purpose:
//    ELITIST stores the best member of the previous generation.
{
  int i;
  double best;
  int best_mem;
  double worst;
  int worst_mem;

  best = fitness[0];
  worst = fitness[0];

  for ( i = 0; i < popsize - 1; ++i ) {
    if ( fitness[i+1] > fitness[i] ) {
      if ( best >= fitness[i] ) {
        best = fitness[i];
        best_mem = i;
      }
      if ( fitness[i+1] >= worst ) {
        worst = fitness[i+1];
        worst_mem = i + 1;
      }
    } else {

      if ( fitness[i] >= worst ) {
        worst = fitness[i];
        worst_mem = i;
      }

      if ( best >= fitness[i+1] ) {
        best = fitness[i+1];
        best_mem = i + 1;
      }
    }
  }
```

```
//  If the best individual from the new population is better than
//  the best individual from the previous population, then
//  copy the best from the new population; else replace the
//  worst individual from the current population with the
//  best one from the previous generation
//
  if ( fitness[popsize-1] >= best ) {
    population[popsize-1] = make_shared<GenomeRankedNode>(population[best_mem]);
    fitness[popsize-1] = fitness[best_mem];
  } else {
    population[worst_mem] = make_shared<GenomeRankedNode>(population[popsize-1]);
    fitness[worst_mem] = fitness[popsize-1];
  }

  return;
}

void GeneticAlgorithm::evaluate ( )
//  Purpose:
//    EVALUATE implements the user-defined valuation function
{
  int member;
  for ( member = 0; member < popsize; member++ ) {
      fitness[member] = brierScore->calculateBrierScore(population[member]);
  }
  return;
}

int GeneticAlgorithm::i4_uniform_ab ( int a, int b, int &seed )
//  Purpose:
//    I4_UNIFORM_AB returns a scaled pseudorandom I4 between A and B.
{
  int c;
  const int i4_huge = 2147483647;
  int k;
  float r;
  int value;
  if ( seed == 0 ) {
    cerr << "\n";
    cerr << "I4_UNIFORM_AB - Fatal error!\n";
    cerr << "  Input value of SEED = 0.\n";
    exit ( 1 );
  }

  //  Guarantee A <= B.
  if ( b < a ) {
    c = a;
    a = b;
    b = c;
  }

  k = seed / 127773;
  seed = 16807 * ( seed - k * 127773 ) - k * 2836;
  if ( seed < 0 ) {
    seed = seed + i4_huge;
  }

  r = ( float ) ( seed ) * 4.656612875E-10;
    //  Scale R to lie between A-0.5 and B+0.5.
  r = ( 1.0 - r ) * ( ( float ) a - 0.5 )
    +         r   * ( ( float ) b + 0.5 );

  //  Use rounding to convert R to an integer between A and B.
  value = round ( r );
  //  Guarantee A <= VALUE <= B.
  if ( value < a ) value = a;
  if ( b < value ) value = b;

  return value;
}

double GeneticAlgorithm::round(double number) {
    return number < 0.0 ? ceil(number - 0.5) : floor(number + 0.5);
}
```

```
void GeneticAlgorithm::initialize () {
//  Create an initial random population of ranked nodes
    int numParents = brierScore->getNumParents();

    for (int i = 0; i < popsize; i++ ) {
        population.push_back(RandomRankedTPNGenerator::randomNode(numParents));
        fitness.push_back(0);
    }
    return;
}

void GeneticAlgorithm::keep_the_best ( )
//  Purpose:
//    KEEP_THE_BEST keeps track of the best member of the population.
{
  int cur_best;
  int mem;

  cur_best = 0;
  for ( mem = 0; mem < popsize; mem++ ) {
      if ( fitness[popsize-1] > fitness[mem] ) {
      cur_best = mem;
      fitness[popsize-1] = fitness[mem];
    }
  }
  //Once the best member in the population is found, copy the genes.
  population[popsize-1] = population[cur_best];
  return;
}

void GeneticAlgorithm::mutate ( int &seed )
//  Purpose:
//    MUTATE performs a random uniform mutation.
{
  const double a = 0.0;
  const double b = 1.0;
  int i;
  double x;
  for ( i = 0; i < popsize; i++ )
  {
      x = r8_uniform_ab ( a, b, seed );
      if ( x < pmutation )
      {
        // get the parameters of the current gene
        int currentFunction = population[i]->getFunction();
        int* currentWeights = population[i]->getWeights();
        int currentVariance = population[i]->getVariance();
        int numParents = population[i]->getNumParents();

        //Randomly decide which parameter to mutate
        int pos = RandomRankedTPNGenerator::randomr(0,2);

        // Perform a mutation given the randomly generated number (pos)
        if (pos == 0) {
            currentFunction = RandomRankedTPNGenerator::randomFunction();
        } else if (pos == 1) {
            currentWeights = RandomRankedTPNGenerator::randomWeights(numParents);
        } else {
            currentVariance = RandomRankedTPNGenerator::randomVariance();
        }
        population[i] = make_shared<GenomeRankedNode>(currentFunction, numParents, currentWeights, currentVariance);
      }
  }
}

double GeneticAlgorithm::r8_uniform_ab ( double a, double b, int &seed )
//  Purpose:
//    R8_UNIFORM_AB returns a scaled pseudorandom R8.
{
  int i4_huge = 2147483647;
  int k;
  double value;
```

```
  if ( seed == 0 ) {
    cerr << "\n";
    cerr << "R8_UNIFORM_AB - Fatal error!\n";
    cerr << "  Input value of SEED = 0.\n";
    exit ( 1 );
  }

  k = seed / 127773;
  seed = 16807 * ( seed - k * 127773 ) - k * 2836;

  if ( seed < 0 ) seed = seed + i4_huge;
  value = ( double ) ( seed ) * 4.656612875E-10;
  value = a + ( b - a ) * value;
  return value;
}

void GeneticAlgorithm::report ( int generation )
//  Purpose:
//    REPORT reports progress of the simulation.
{
  double avg, best_val, square_sum, stddev, sum, sum_square;
  int i;

  if ( generation == 0 ) {
    std::cout << "\n";
    std::cout << "  Generation       Best            Average       Standard \n";
    std::cout << "  number           value           fitness       deviation \n";
    std::cout << "\n";
  }

  sum = 0.0;
  sum_square = 0.0;
  for ( i = 0; i < popsize; i++ ) {
    sum = sum + fitness[i];
    sum_square = sum_square + fitness[i] * fitness[i];
  }
  avg = sum / ( double ) popsize;
  square_sum = avg * avg * popsize;
  stddev = sqrt ( ( sum_square - square_sum ) / ( popsize - 1 ) );
  best_val = fitness[popsize-1];
  std::cout << "  " << setw(8) << generation
       << "  " << setw(14) << best_val
       << "  " << setw(14) << avg
       << "  " << setw(14) << stddev << "\n";

  return;
}

void GeneticAlgorithm::selector ( int &seed )
//  Purpose:
//    SELECTOR is the selection function.
{
  const double a = 0.0, b = 1.0;
  int i, j, mem;
  double p, sum;

  //  Find the total fitness of the population.
  sum = 0.0;
  for ( mem = 0; mem < popsize; mem++ ) {
    sum = sum + fitness[mem];
  }

  //  Calculate the relative fitness of each member.
  for ( mem = 0; mem < popsize; mem++ ) {
    rfitness.push_back(fitness[mem] / sum);
  }

  //  Calculate the cumulative fitness.
  cfitness.push_back(rfitness[0]);
  for ( mem = 1; mem < popsize; mem++ ) {
    cfitness.push_back(cfitness[mem-1] + rfitness[mem]);
  }
```

```
  //  Select survivors using cumulative fitness.
  for ( i = 0; i < popsize; i++ ) {
    p = r8_uniform_ab ( a, b, seed );
    if ( p < cfitness[0] ) {
      newpopulation[i] = population[0];
    } else {
        for ( j = 0; j < popsize; j++ ) {
            if ( cfitness[j] <= p && p < cfitness[j+1] ) {
            newpopulation[i] = population[j+1];
            }
          }
        }
    }

  //  Overwrite the old population with the new one.
  for ( i = 0; i < popsize; i++ ) {
      population[i] = newpopulation[i];
  }

  return;
}
void GeneticAlgorithm::timestamp ( )
//  Purpose:
//    TIMESTAMP prints the current YMDHMS date as a time stamp.
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  size_t len;
  time_t now;

  now = time ( NULL );
  tm = localtime ( &now );
  len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );
  std::cout << time_buffer << "\n";

  return;
# undef TIME_SIZE
}

void GeneticAlgorithm::Xover ( int one, int two, int &seed )
//  Purpose:
//    XOVER performs a two-point crossover of the two selected parents.
{
  //Swap Weights
    int numParents = brierScore->getNumParents();
    int *temp = new int(numParents);

    for (int i = 0; i < numParents; i++)
        temp[i] = population[one]->getWeights()[i];
    population[one]->setWeights(population[two]->getWeights());
    population[two]->setWeights(temp);
  return;
}
```

Figura D.2: geneticAlgorithm.cpp

```cpp
#include "genomeRankedNode.h"
#include <sstream>

double vars[11] = {0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50};

GenomeRankedNode::GenomeRankedNode() {
    function = 0;
    weights = 0;
    numNodeParents = 1;
    variance = 0;
}

GenomeRankedNode::GenomeRankedNode(int inFunction, int inNumParents, int* inWeights, int inVariance) {
    function = inFunction;
    numNodeParents = inNumParents;
    weights = inWeights;
    variance = inVariance;
}

GenomeRankedNode::GenomeRankedNode(GenomeRankedNode* inNode) {
    function = inNode->getFunction();
    numNodeParents = inNode->getNumParents();
    weights = inNode->getWeights();
    variance = inNode->getVariance();
}

GenomeRankedNode::GenomeRankedNode(shared_ptr<GenomeRankedNode> inNode) {
    function = inNode->getFunction();
    numNodeParents = inNode->getNumParents();
    weights = inNode->getWeights();
    variance = inNode->getVariance();
}

GenomeRankedNode::GenomeRankedNode(const GenomeRankedNode &inNode) {
    function = inNode.getFunction();
    numNodeParents = inNode.getNumParents();
    variance = inNode.getVariance();
}

GenomeRankedNode::GenomeRankedNode(const GenomeRankedNode &inNode) {
    function = inNode.getFunction();
    numNodeParents = inNode.getNumParents();
    variance = inNode.getVariance();

    //deep copy
    if (inNode.getWeights()){
        weights = new int[numNodeParents];
        for (int i = 0; i < numNodeParents; i++)
            weights[i] = inNode.getWeights()[i];
    }
}

GenomeRankedNode& GenomeRankedNode::operator= (const GenomeRankedNode &inNode) {

    //self-assignment check
    if (this == &inNode)
        return *this;

    function = inNode.getFunction();
    numNodeParents = inNode.getNumParents();
    variance = inNode.getVariance();

    // explicitly deallocate values
    delete[] weights;

    // deep copy
    if (inNode.getWeights()){
        weights = new int[numNodeParents];
        for (int i = 0; i < numNodeParents; i++)
            weights[i] = inNode.getWeights()[i];
    }
    else
        weights = 0;

    return *this;
}
```

```
72  GenomeRankedNode::~GenomeRankedNode(void) {
73      //delete weights;
74  }
75
76  int GenomeRankedNode::getFunction() const {
77      return function;
78  }
79
80  int* GenomeRankedNode::getWeights() const {
81      return weights;
82  }
83
84  int GenomeRankedNode::getNumParents() const {
85      return numNodeParents;
86  }
87
88  int GenomeRankedNode::getVariance() const {
89      return variance;
90  }
91
92  void GenomeRankedNode::setWeights(int *inWeights) {
93      for (int i = 0; i < numNodeParents ; i++){
94          weights[i] = inWeights[i];
95      }
96  }
97
98  void GenomeRankedNode::setFunction(int inFunction) {
99      function = inFunction;
100 }
101
102 void GenomeRankedNode::setVariance(int inVariance) {
103     variance = inVariance;
104 }
105
106 double GenomeRankedNode::getRealVariance(int inVariance) const {
107     return vars[0];
108 }
    string GenomeRankedNode::toString() {
        std::string str;

        str = "Function: ";
        if (function == 0) str += "WMEAN\n";
        else if (function == 1) str += "WMAX\n";
        else if (function == 2) str += "WMIN\n";
        else str += "MIXMINMAX\n";
        str += "Weights: ";
        for (int i = 0; i < numNodeParents; i++){
            std::ostringstream oss;
            oss << weights[i];
            str += oss.str() + " ";
        }
        str += "\n";
        std::ostringstream oss;
        oss << variance;
        str += "Variance: " + oss.str();
        str += "\n";

        return str;
    }
```

Figura D.3: genomeRankedNode.cpp

```cpp
#include "randomRankedTPNGenerator.h"
#include "genomeRankedNode.h"
#include <stdlib.h>
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <fstream>
# include <iomanip>
# include <cmath>
# include <ctime>
# include <cstring>

using namespace std;

int minimumWeight = 1;
int maximumWeight = 5;

RandomRankedTPNGenerator::RandomRankedTPNGenerator(void) {}

int RandomRankedTPNGenerator::getMinWeight() {
    return minimumWeight;
}

int RandomRankedTPNGenerator::getMaxWeight() {
    return maximumWeight;
}

void RandomRankedTPNGenerator::setMinWeight(int inMinWeight) {
    minimumWeight = inMinWeight;
}

void RandomRankedTPNGenerator::setMaxWeight(int inMaxWeight) {
    maximumWeight = inMaxWeight;
}
shared_ptr<GenomeRankedNode> RandomRankedTPNGenerator::randomNode(int numParents) {
    int function = randomFunction();
    int* weights = randomWeights(numParents);
    int variance = randomVariance();
    return make_shared<GenomeRankedNode>(function, numParents, weights, variance);
}

// Assumes four types of function
int RandomRankedTPNGenerator::randomFunction() {
    return randomr(0,3);
}

// Assumes 11 possible variances' values
int RandomRankedTPNGenerator::randomVariance() {
    return randomr(0,10);
}

int* RandomRankedTPNGenerator::randomWeights(int numParents) {
    int *weights = new int(numParents);
    for (int i = 0; i < numParents; i++)
        weights[i] = randomr(minimumWeight, maximumWeight);
    return weights;
}

int RandomRankedTPNGenerator::randomr(unsigned int min, unsigned int max) {
    int randNum = rand()%(max-min + 1) + min;
    return randNum;
}

RandomRankedTPNGenerator::~RandomRankedTPNGenerator(void) {}
```

Figura D.4: randomRankedTPNGenerator.cpp